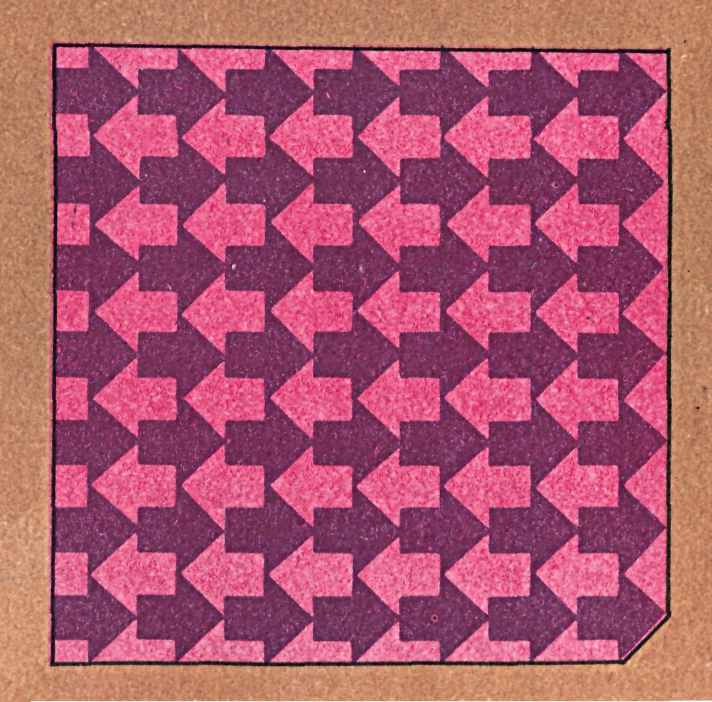
МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Н. Вирт

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ МОДУЛА-2







TEXT AND MONOGRAPHS IN COMPUTER SCIENCE

PROGRAMMING IN MODULA-2

NIKLAUS WIRTH

Third, Corrected Edition

Springer-Verlag
Berlin Heidelberg New York Tokyo
1985

МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ ЭВМ

Н. Вирт

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ МОДУЛА-2

Перевод с английского
В.А. Серебрякова и В.М. Ходукина
под редакцией
В.М. Курочкина



МОСКВА «МИР» 1987

ББК 32.973
В 52
УДК 681.3

Вирт Н.
В52 Программирование на языке Модула-2: Пер. с англ. — М.: Мир, 1987. — 224 с., ил.

Книга известного швейцарского специалиста по системному программированию, знакомого советским читателям по переводам его книг «Введение в системное программирование» (М.: Мир, 1977) и «Алгоритмы + структуры данных = программы» (М.: Мир, 1985). Язык Модула-2 является преемником известного языка Паскаль и ориентирован на однопроцессорные малые ЭВМ. Книга сочетает в себе достоинства учебного пособия и справочного руководства по этому языку.

Для системных программистов, для специалистов, работающих с языком Модула-2.

В $\frac{1702070000-045}{041(01)-88}$ 102 — 88, ч.I ББК 32.973

Редакция литературы по математическим наукам

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Более 15 лет назад появился язык Паскаль, который быстро завоевал популярность, получив широкое распространение. Создан он был для целей обучения программированию, однако очень скоро нашел другое поприще – системное программирование. И пожалуй, в дальнейшем (в том числе и у нас) он больше всего используется именно для создания программного обеспечения. Правда, целый ряд черт языка мешал этому. Прежде всего отсутствовала модульность. Немалую роль играли также такие моменты, как отсутствие в языке параллельных процессов, затруднения с организацией работы различных независимых устройств вычислительной машины и др. Видимо, все это привело Н. Вирта к идее разработки языка Модула, а затем и настоящей его модификации – Модула-2. С того момента, когда этот язык стал известен системным программистам в СССР, число его сторонников постоянно увеличивалось. Можно смело рассчитывать на то, что в ближайшее время, кроме зарубежных, мы будем располагать целым рядом высококачественных отечественных трансляторов, в большей степени приспособленных к нашей специфике работы на ЭВМ.

Трудно объяснить причины успеха (или неуспеха) какого-либо языка программирования. Помимо привычки (возможно, основной причины, объясняющей распространенность, например, Фортрана), есть еще какие-то факторы. И не исключено, что весьма существенным для языков, создаваемых Н. Виртом, и в частности для Модулы-2, является относительная простота: при всей широте возможностей и мощности изобразительных средств описание его требует всего 40 страниц ("Сообщение о языке программирования Модула-2" в настоящей книге). Это, конечно, существенно облегчает изучение языка и его использование.

Следует все же сказать, что в языке не все может нравиться. В таких оценках много субъективного, но тем не менее хочется отметить, например, отсутствие динамических массивов, бедность аппарата параллельных процессов и средств их взаимодействия, отсутствие способов гибкого задания отображения типов на физическую память машины. Создалось впечатление, что сложные системные программы будут ориентированы на те ЭВМ, для которых они пишутся, и перенос программ с одних ЭВМ на другие будет затруднен. Впрочем, язык Модула-2 не следует рассматривать как окончательно сформированный и законченный, и возможно, что как

по инициативе самого Н. Вирта, так и в результате накопления опыта работы с языком в нем будут происходить изменения. Это, в частности, подтверждается дрейфом языка, наблюдаемым в различных авторских публикациях описаний языка. В частности, в 1985 г. появился препринт (**N. Wirth. A fast and compact compiler for Modula-2. J. Gutknecht. Compilation of data structures: an new approach to efficient Modula-2 symbol files. July 1985, #64, Institut fur informatic, ETH-Zentrum, Zurich, Switzerland**), в котором Н. Вирт требует объявления объектов (констант, переменных, процедур) до их использования. Для процедур разрешено предварительное описание заголовков.

В настоящем переводе, выполнявшемся с третьего английского издания, часть идентификаторов в программах не переведена на русский язык. Неизменными остались идентификаторы в тех модулях, которые могут войти в библиотеки Модулы-2 в качестве стандартных. При подготовке русского издания переводчики и редакторы пользовались средствами современной вычислительной техники.

Будем надеяться, что книга окажется очень интересной для советских читателей и принесет большую пользу, в первую очередь разработчикам программного обеспечения. Кроме того, первая (и основная) часть книги, задуманная скорее как введение в язык программирования Модула-2, а не как его строгое определение, может служить прекрасным учебником по программированию, написанным свойственным Н. Вирту четким языком, выдержанным в стиле структурного программирования и иллюстрированным весьма интересными примерами.

В. М. Курочкин

ПРЕДИСЛОВИЕ

Настоящая книга представляет собой введение в программирование вообще и руководство по программированию на языке Модула-2 в частности. Она ориентирована в основном на лиц, уже знакомых с элементами программирования и желающих систематизировать свои знания в этой области. Тем не менее в книгу включен вводный раздел для начинающих, где в сжатом виде представлены фундаментальные понятия информатики, благодаря чему книга может служить и самоучителем. Используемая здесь система обозначений – это язык Модула-2, которому в большой мере присущ структурный подход. Он вырабатывает у изучающего стиль работы, широко известный под названием структурное программирование.

Эта книга, служащая руководством по программированию на языке Модула-2, охватывает практически все его средства. В гл. 1 рассматриваются такие основные понятия, как переменная, выражение, присваивание, условный оператор, оператор цикла, а также массивы. Эта и вторая глава, в которой вводится важное понятие процедуры или подпрограммы, по существу, содержат материал стандартного вводного курса программирования. Глава 3 касается типов и структур данных, что составляет ядро курсов программирования повышенного типа. Четвертая глава посвящена понятию модуля, являющегося фундаментальным средством при разработке больших программных систем и при совместной работе коллективов программистов. Наиболее широко используемые служебные программы ввода и вывода даны в виде примеров модулей. И наконец, в гл. 5 описываются средства системного программирования, работа с внешними устройствами и мультипрограммирование. Книга содержит практические рекомендации по тому, как и где использовать конкретные средства языка. Эти рекомендации должны помочь читателю выработать хороший стиль программирования.

Язык Модула-2 – потомок и прямой наследник языков Паскаль [1] и Модула [2]. Паскаль был разработан как язык общего назначения и после его реализации в 1970 г. получил широкое распространение, а Модула возникла из экспериментов по мультипрограммированию и нацелена, следовательно, на аспекты, относящиеся именно к этой сфере приложений. Язык Модула был специфицирован и реализован в опытном порядке в 1975 г.

В 1977 г. в Цюрихе, в Институте информатики (Institut fur

Informatic of ETH, Zurich) была начата работа по созданию новой вычислительной системы. Проект предполагал одновременную разработку аппаратуры и программного обеспечения. Эта система (позже названная Lilith) должна была программироваться на едином языке высокого уровня, который, следовательно, должен был, с одной стороны, удовлетворять требованиям проектирования в целом, а с другой – допускать программирование ее отдельных фрагментов, описывающих взаимодействие с аппаратурой. В результате скрупулезного анализа проекта возник язык Модула-2, включающий все характерные черты Паскаля и дополненный важными понятиями модуля и мультипрограммирования. Поскольку синтаксис нового языка соответствовал больше синтаксису Модулы, чем Паскаля, было выбрано название Модула-2. Далее мы будем использовать названия Модула и Модула-2 как синонимы.

От Паскаля язык отличается следующими основными средствами:

1. Понятие модуля и возможность его разбиения на раздел определений и раздел реализации.
2. Более систематизированный синтаксис, что облегчает изучение языка. В частности, каждая конструкция, начинающаяся с ключевого слова, заканчивается тоже ключевым словом (за исключением оператора REPEAT ... UNTIL ...), т.е. заключена в своего рода скобки.
3. Процесс – как ключевое понятие мультипрограммирования.
4. Так называемые средства программирования низкого уровня, позволяющие ослабить жесткий контроль типов и отображать данные, имеющие структуру Модулы-2, на память, не обладающую внутренней структурой.
5. Процедурный тип, который позволяет динамически присваивать процедуры переменным.

Первая реализация Модулы-2 заработала на PDP-11 в 1979 г., а первое определение языка было опубликовано в марте 1980 г. как сообщение о языке Института информатики. С тех пор язык интенсивно используется в стенах нашего института. После годичной эксплуатации и проверок на различных приложениях в марте 1981 г. компилятор был передан внешним пользователям. Интерес к компилятору быстро возрос, поскольку он оказался мощным инструментом разработки сложных систем и был реализован на широко распространенной мини-ЭВМ. Этот интерес вызвал необходимость написания руководства и учебника по языку. Сообщение о языке, содержащее сжатое определение языка Модула-2, включено в конец настоящего руководства в основном для облегчения ссылок на него. Оно осталось практически неизменным; в нем лишь опущены разделы, посвященные стандартным служебным модулям и использованию компилятора.

Английский оригинал книги был получен в виде, удобном для тиражирования, с помощью мини-ЭВМ Lilith, присоединенной к лазерному печатающему устройству Canon LBP-10. Параллельно с написанием книги автор разрабатывал программы, необходимые для форматирования текстов (и управления печатающим устройством), а также проектировал интерфейс связи с устройством печати. Естественно, что все эти программы были написаны на Модуле (для Lilith).

Просто невозможно выразить заслуженную благодарность всем, кто оказал влияние на написание этой книги и на проект Модула-2. Большую пользу принес мне год (1976), проведенный в исследовательской лаборатории корпорации Xerox, и знакомство с некоторыми идеями, касающимися модульного программирования, содержащимися в языке Mesa [3]. Вероятно, очень важной была мысль о возможности эффективной реализации языка высокого уровня на мини-ЭВМ. Приношу свою благодарность также разработчикам Модулы, в особенности Л.Гайсмену, А.Горангуру, Ч.Якоби и С.Е.Кнудсену, которые не только превратили Модулу в эффективный и надежный инструмент, но также часто (и очень мудро) предостерегали меня от включения в язык новых модных средств.

ПРЕДИСЛОВИЕ К ТРЕТЬЕМУ ИЗДАНИЮ

В третье издание книги включены изменения и модификации Модулы-2, сделанные в конце 1983 г. Изменения по сравнению с предыдущими изданиями отмечены символом (!). Одно существенное изменение касается модуля определений, который теперь не содержит экспортного списка, а сам фактически представляет собой экспортный список (см. разд. 24). Дополнительно в приложение включены несколько стандартных модулей, оказавшихся весьма полезными. В основном они относятся к вводу и выводу, т.е. использованию клавиатуры, дисплея и файловой системы.

Н.Вирт

Литература

1. N.Wirth. The programming language PASCAL. Acta Informatica 1, 35-63 (1971).
2. N.Wirth. Modula: a language for modular multiprogramming. Software – Practice and Experimence,7,3-35 (1977).
3. J.G.Mitchel, W.Maybury, R.Sweet. Mesa Language Manual. Xerox PARC, CSL-78-1 (1978).

Часть 1

1. ВВЕДЕНИЕ

Хотя данное руководство и предполагает знакомство читателя с основными понятиями информатики, однако, по-видимому, все же уместно начать с объяснения некоторых понятий и терминологии. Мы осознаем, что (за редким исключением) программы пишутся (более точно – проектируются) с целью их интерпретации вычислительной машиной. Машина в этом случае выполняет некий процесс, т.е. последовательность действий в соответствии со спецификацией, заданной программой. Этот процесс также называется вычислением.

Программа сама по себе – это не что иное, как текст. Поскольку она, как правило, определяет достаточно сложный процесс и должна делать это с максимальной точностью и учетом всех деталей, смысл этого текста должен быть определен очень строго. Такая строгость требует наличия некоторого формализма, для которого теперь используется термин язык. Мы принимаем это название, хотя на языке обычно говорят, и он гораздо менее четко определен. Наша цель здесь – изучить формализм, или язык, называемый Модула-2 (далее – просто Модула).

Программа обычно определяет процесс, который заставляет интерпретатор, т.е. ЭВМ, считывать данные (так называемый ввод) из некоторых источников и варьировать свои последующие действия в зависимости от вводимых данных. Имеется в виду, что программой определяется не единственный процесс, а целый класс вычислений (обычно неограниченный). Мы должны гарантировать, что во всех случаях эти процессы будут действовать в соответствии с заданными описаниями (или, следовало бы сказать, нашими ожиданиями). Мы могли бы проверить, что описание действительно удовлетворяется в случае единственного процесса вычислений, но в общем случае это невозможно, поскольку класс всех допустимых процессов слишком велик. Добросовестный программист обеспечивает правильность своей программы путем ее тщательной разработки и анализа. Именно тщательная разработка – сущность профессионального программирования.

Задача проектирования программы еще больше осложняется тем, что программа должна не только полностью описывать класс вычислений, а часто также должна интерпретироваться (выполняться) различными интерпретаторами (вычислительными

машинами). Раньше это требовало ручного перевода исходного вида программы в различные машинные коды, причем приходилось принимать во внимание разнообразные характеристики и ограничения этих кодов. С созданием языков высокого уровня, имеющих формальные определения, и построением автоматических трансляторов, преобразующих программы в коды различных машин, трудности коренным образом уменьшились, хотя и не исчезли. В принципе формальный язык следовало бы определять абстрактным образом, возможно аксиоматически, не делая никаких ссылок на реальную машину или на конкретный механизм интерпретации. Если бы это было достигнуто, то программисту нужно было понимать только сам формальный язык. Однако такая общность часто слишком сильно ограничивает действия программиста и требует дополнительных затрат, поэтому во многих случаях он все же должен знать основные характеристики своей машины или машин. Тем не менее квалифицированный программист будет стремиться как можно меньше обращаться к специфическим характеристикам машины, а опираться исключительно на стандарт формального языка, чтобы сохранить универсальность и мобильность программы. Язык Модула помогает решать эту задачу: машинные зависимости заключаются в особые объекты, используемые только в так называемом низкоуровневом программировании.

Из сказанного выше становится ясно, что процесс трансляции расположен между написанием программы и ее интерпретацией. Процесс этот называется компиляцией и состоит в переводе исходного текста в кодированное машинное представление. Качество компиляции может оказывать решающее влияние на эффективность последующей интерпретации программы. Мы подчеркиваем тот факт, что может быть много компиляторов с одного языка (даже для одной ЭВМ): одни более, другие менее эффективные. Важно понимать, что эффективность – характеристика реализации, а не языка, и, следовательно, необходимо разделить понятия "язык" и "реализация".

Подведем итог:

Программа – фрагмент текста.
Она задает процесс вычислений.
Процесс осуществляется некоторым интерпретатором, обычно вычислительной машиной, интерпретирующей (выполняющей) программу.
Смысл программы задается формализмом, называемым языком программирования.
Программа задает некоторый класс вычислений, причем исходные данные играют роль параметра каждого конкретного процесса.
Перед выполнением текст программы транслируется компилятором в машинный код. Этот процесс называется компиляцией.

Разработка программы включает в себя обеспечение того, чтобы все члены упомянутого класса вычислений функционировали в соответствии с определением. Это осуществляется тщательной аналитической верификацией программы и избирательным тестированием характерных вариантов.

В программах по возможности следует воздерживаться от использования особенностей конкретных интерпретаторов (вычислительных машин). Только в этом случае можно быть уверенным, что смысл программы будет понят по описанию языка.

Компилятор – программа, переводящая исходный вид программы в коды конкретной машины. Перед выполнением программа должна быть откомпилирована. Программирование в широком смысле слова подразумевает не только составление программы, но также и конкретную подготовку текста, его компиляцию, исправление ошибок, так называемую отладку и планирование тестов. Современный программист использует для этих целей много различных средств, включая редакторы, компиляторы и отладчики. Он также должен быть знаком с программным окружением этих компонент. Мы не будем касаться всех этих аспектов, а сосредоточим внимание на языке Модула.

2. ПЕРВЫЙ ПРИМЕР

Проследим этапы разработки простой программы и поясним на ее примере некоторые фундаментальные понятия программирования и основные средства языка Модула. Рассмотрим следующую задачу: даны два натуральных числа x и y; надо вычислить их наибольший общий делитель (нод).

Приведем необходимые для решения этой задачи математические сведения.

1. Если x равен y, то x (или y) – искомый результат.
2. нод двух чисел не изменится, если большее из них заменить их разностью, т.е. вычесть из большего числа меньшее.

Если выразить это в математических терминах, то получим следующие правила:

1. нод(x,x) = x
2. Если x > y, то нод(x,y) = нод(x–y,y)

Основной рецепт, так называемый алгоритм получения нод, таков: изменять числа x и y согласно правилу 2 так, чтобы их разность уменьшалась. Повторять это до тех пор, пока числа не станут равными. Правило 2 гарантирует, что при этих изменениях нод(x,y) все время остается одним и тем же, а правило 1 гарантирует, что в конце концов мы найдем результат.

Теперь мы должны записать эти рекомендации в терминах Модулы. Первая попытка приводит к следующему наброску (первая версия). Заметим, что символ # означает "не равно".

```
WHILE x # y DO
 "применить правило 2, уменьшив разность"
END
```

Текст в кавычках представляет собой предложение естественного языка. Вторая версия уточняет первую, заменяя естественный язык формальными терминами.

```
WHILE x # y DO
 IF x > y THEN
  x := x - y
 ELSE
  y := y - x
 END
END
```

Этот фрагмент текста – еще не готовая программа, но он уже демонстрирует одну существенную черту языка структурного программирования – иерархическую структуру. Вся первая версия – это один оператор, и он содержит другой подчиненный "оператор" (текст в кавычках). Во второй версии этот внутренний "оператор" детализирован, и появились новые подчиненные операторы, заменяющие значение x значением x – y. Такая иерархия операторов отражает структуру, лежащую в основе алгоритма. Она явно видна, благодаря структуре языка, разрешающего вложение компонент программы друг в друга. Поэтому важно знать структуру, т. е. синтаксис языка до самых мельчайших деталей. В тексте мы отразили вложение или подчинение сдвигами строк. Хотя это и не требуется нормами языка, но существенно помогает пониманию программы.

Отражение внутренней структуры алгоритма в структуре текста программы – ключевая идея структурного программирования. По существу, невозможно понять смысл программы, если исчезнет ее структура, как это бывает, когда компилятор выдает машинный код. Мы должны иметь в виду следующее – программа бесполезна, если человек не может в ней разобраться и удостовериться в ее правильности.

Теперь приступим к получению из написанного выше фрагмента законченной программы. Понятно, что нужно задать действия, присваивающие начальные значения переменным x и y, и действие, делающее видимым результат. Казалось бы, для этой цели нам потребуется знание машинных средств связи с пользователем. Но поскольку мы не хотим обращаться к специфике конкретных машин, особенно в таком важном и часто встречающемся случае, как генерация выходной информации, введем абстракции средств связи,

предполагая, что они будут в наличии (реализованы некоторым подходящим образом) на всех ЭВМ, на которых возможно программирование на Модуле. Эти абстракции, как показано ниже, принимают форму стандартных операторов. Ввод данных осуществляется операцией Read (читать), а вывод – операцией Write (писать). Мы можем, например, считать, что данные читаются с клавиатуры и пишутся на дисплей.

```
ReadCard(x);
ReadCard(y);
WHILE x # y DO
 IF x > y THEN x := x - y
        ELSE y := y - x
 END
END;
WriteCard(x,6)
```

Процедура ReadCard читает число типа CARDINAL (т.е. целое неотрицательное) и присваивает его параметру (x). Процедура WriteCard выводит число типа CARDINAL, указанное ее первым параметром (x). Второй параметр (6) указывает количество позиций, выделяемое для представления этой величины на внешнем носителе. В следующей далее окончательной версии мы оформим наш текст так, что он станет настоящей программой на Модуле.

```
MODULE нод;
 FROM InOut IMPORT ReadCard,WriteString,
                   WriteLn,WriteCard;
 VAR x,y: CARDINAL;
BEGIN
 WriteString("x="); ReadCard(x); WriteLn;
 WriteString("y="); ReadCard(y); WriteLn;
 WHILE x # y DO
  IF x > y THEN x := x - y
        ELSE y := y - x
  END
 END;
 WriteString("нод="); WriteCard(x,6); WriteLn;
END нод.
```

Существенные добавления, сделанные на этом шаге, – это описания. В Модуле имена всех объектов, встречающихся в программе, таких, как переменные и константы, должны быть описаны. Описание вводит идентификатор (имя) объекта, определяет вид объекта (переменная, константа или что-либо еще) и указывает его общие неизменные свойства, такие, как тип переменной или значение константы.

Получившейся завершенной программе, называемой модулем, присваивается имя (нод), и она имеет следующий формат:


```
MODULE имя:
 <списки импорта>
 <описания>
BEGIN
 <операторы>
END имя.
```

Уместно сделать еще несколько замечаний относительно нашего примера. Процедуры WriteLn, WriteString, ReadCard и WriteCard не являются частью самого языка. Они определены в другом модуле, называемом InOut, который считается доступным. Подборка таких полезных модулей будет приведена в последующих разделах книги с соответствующими пояснениями. Здесь же мы просто отметим: для того чтобы сделать модули доступными программе, их нужно импортировать. Это осуществляется включением имен нужных объектов в список импорта и указанием того, какому модулю они принадлежат.

Процедура WriteString выводит текст в виде последовательной цепочки литер (предназначенные для вывода литеры заключены в кавычки). Этот вывод сообщает пользователю ЭВМ, что далее требуется ввод. Такое поведение – существенное свойство диалоговых систем. Процедура WriteLn заканчивает строку в выходном тексте.

На этом завершим обсуждение первого примера, которое было совершенно неформальным. Это допустимо, поскольку мы стремились объяснить уже готовую программу. Однако программирование – это разработка и создание новых программ. Для такой цели подходит лишь точное, формальное описание нашего инструментального средства. В следующем разделе мы введем формализм для точного описания правильных ("законных") программ. Этот формализм позволяет строгим образом определить, удовлетворяет ли написанный текст нормам языка.

3. НОТАЦИЯ ДЛЯ ЗАПИСИ СИНТАКСИСА МОДУЛЫ

Формальный язык – бесконечное множество цепочек символов. Элементы этого множества называются предложениями языка. В случае языка программирования такими предложениями являются программы. Символы берутся из конечного множества, называемого словарем. Так как множество программ бесконечно и не может быть задано прямым перечислением, то вместо этого оно определяется правилами образования его элементов. Последовательности символов, которые могут быть образованы в соответствии с этими правилами, называют синтаксически правильными программами. Такой набор правил представляет собой синтаксис языка.

Программы формального языка соответствуют грамматически правильным предложениям разговорных языков. Каждое предложение имеет структуру и состоит из отдельных частей, таких, как

подлежащее, сказуемое, дополнение. Аналогично, программа состоит из частей, называемых синтаксическими понятиями, таких, как операторы, выражения, описания. Если грамматическая конструкция A состоит из следующих друг за другом конструкций B и C, т.е. их конкатенации (сцепления) BC, то мы будем называть B и C – синтаксическими факторами и описывать A следующей синтаксической формулой:

A = BC.

Если же A состоит либо из B, либо из C, мы будем называть B и C синтаксическими термами и выражать A в виде:

A = B|C.

Для группировки термов и факторов можно использовать круглые скобки. Следует заметить, что A, B и C обозначают синтаксические понятия описываемого формального языка, символы равно "=", вертикальная черта "|", скобки "(",")" и точка "." – символы метанотации, называемые метасимволами. Введенная здесь метанотация называется расширенной формой Бэкуса-Наура (РБНФ).

Кроме конкатенации и выбора РБНФ позволяет выразить условное вхождение и повторение. Если конструкция A может состоять либо из B, либо из пустой цепочки, то это выражается в виде

A = [B].

Если же A может состоять из конкатенации любого числа (включая нуль) конструкций B, то это обозначается

A = {B}.

Вот мы и объяснили, что такое РБНФ. Приведем несколько примеров того, как множества предложений описываются формулами в РБНФ.

(A|B)(C|D)	AC AD BC BD
A[B]C	ABC AC
A{BA}	A ABA ABABA ABABABA ...
{A|B}C	C AC BC AAC ABC BBC BAC ...

Очевидно, что сама РБНФ – это тоже формальный язык. Если этот язык способен делать то, для чего предназначен (описывать формальные языки), то уж по крайней мере он должен уметь описать сам себя! В приведенном ниже описании РБНФ на РБНФ мы используем следующие имена для синтаксических понятий:

СинтОператор	: синтаксическая формула
СинтВыражение	: список альтернативных термов
СинТерм	: конкатенация факторов
СинтФактор	: единичное синтаксическое понятие или выражение в скобках

Формальное определение РБНФ теперь можно задать следующим образом:

```
Синтаксис     = { СинтОператор }.
СинтОператор  = идентификатор "=" СинтВыражение ".".
СинтВыражение = СинТерм { "|" СинТерм }.
СинТерм       = СинтФактор { СинтФактор }.
СинтФактор = идентификатор | цепочка
     | "(" СинтВыражение ")" | "[" СинтВыражение "]"
     | "{" СинтВыражение "}".
```

Идентификаторами обозначены синтаксические понятия; цепочка – это последовательность литер, взятых из алфавита определяемого языка. Для представления идентификатора мы приняли широко используемое в языках программирования соглашение, а именно:

Идентификатор состоит из последовательности букв и цифр, начинающейся буквой. Цепочка состоит из последовательности любых литер, заключенных в кавычки (или в апострофы).

Формальное определение этих правил в терминах РБНФ дано в следующем разделе.

4. ПРЕДСТАВЛЕНИЕ ПРОГРАММ НА МОДУЛЕ

В предыдущем разделе был введен формализм, которым в дальнейшем будет определяться структура правильно составленных программ. Этот формализм, однако, определяет только то, каким образом представляются программы в виде последовательностей синтаксических элементов (лексем), но не литер. Этот "дефект" допущен намеренно: мы полагаем, что представление лексем (а значит, и программ) в терминах литер слишком сильно зависит от конкретной реализации, а для определения языка необходим более высокий уровень абстракции. Создание промежуточного уровня представления через последовательности лексем обеспечивает удобную развязку между языком и окончательным представлением программы, которое зависит от доступного набора литер. Как следствие этого мы должны принять ряд правил, регулирующих представление лексем в виде последовательности литер. Лексемы Модулы разделяются на следующие классы:

идентификаторы, числа, цепочки, операции, разделители, комментарии.

Правила, регулирующие их представление в терминах стандартного набора литер ISO, следующие:

1. Идентификаторы – последовательности букв (* в оригинале только буквы латинского алфавита, в русском переводе книги используются также буквы русского алфавита, прописные и строчные. – Прим. перев.*) и цифр, начинающиеся с буквы:

$ Идентификатор = Буква {Буква|Цифра}.

Вот примеры правильно составленных идентификаторов:

Алиса **hello** ЧернаяПтица операторWHILE SR71

Примеры слов, не являющихся идентификаторами:

Модула 2	(пробел недопустим)
Модула-2	(содержит дефис)
2N	(первой литерой должна быть буква)
D'Alembert	(содержит апостроф)

Прописные и строчные буквы считаются различными.

Иногда идентификатор (например i) должен быть квалифицирован (уточнен) другим идентификатором (J). Это выражается в том, что перед i размещается J и они разделяются точкой (J.i). Такой объединенный идентификатор называется квалифицированным (сокращенно КвалИдент). Его синтаксис:

$ КвалИдент = {Идентификатор "."} Идентификатор.

2. Числа могут быть целыми или действительными. Целые представляются последовательностями цифр. Действительные числа содержат десятичную точку и дробную часть. Кроме того, в действительном числе может присутствовать порядок. Он задается буквой E (прописная латинская) и произносится как "умножить на десять в степени". Числа не должны содержать пробелов. Примеры правильно записанных чисел:

1981 1 3.25 5.1E3 4.0E-10

А вот примеры последовательностей литер, которые не распознаются как числа:

1,5	запятая в числе недопустима
1'000'000	не может быть апострофов
3.5En	запрещены буквы в числе (за исключением E)

Точные правила образования чисел задаются следующим синтаксисом:

$ Число = Целое | Действительное.
$ Целое = Цифра {Цифра}.
$ Действительное = Цифра{Цифра}"."{Цифра}[Порядок].
$ Порядок = "E" ["+"|"-"] Цифра {Цифра}.

ПРИМЕЧАНИЕ: Если за целым числом следует латинская буква B, то оно воспринимается как восьмеричное, если же за ним следует латинская буква H, то как шестнадцатеричное.

3. Цепочка – последовательность любых литер, заключенная в кавычки. Очевидно, что для однозначного распознавания цепочки необходимо, чтобы она сама не содержала кавычек. Чтобы можно было записать и цепочку, содержащую кавычки, разрешается заключать ее вместо кавычек в апострофы. Однако в этом случае в ней не должно содержаться апострофов.

$ Цепочка = '"'{Литера}'"'|"'"{Литера}"'".

Примеры цепочек:

"NO PROBLEM"
"L'Etoile"
'Придворная молочница сказала: "Благодарствуйте!".'

4. Операции и ограничители – это или специальные литеры или ключевые слова. Последние пишутся прописными буквами и не должны использоваться в качестве идентификаторов. Стоит поэтому запомнить ключевые слова, перечисленые в следующем далее списке: их смысл будет объяснен в последующих разделах.

AND	ELSIF	LOOP	REPEAT
ARRAY	END	MOD	RETURN
BEGIN	EXIT	MODULE	SET
BY	EXPORT	NOT	THEN
CASE	FOR	OF	TO
CONST	FROM	OR	TYPE
DEFINITION	IF	POINTER	UNTIL
DIV	IMPLEMENTATION	PROCEDURE	VAR
DO	IMPORT	QUALIFIED	WHILE
ELSE	IN	RECORD	WITH

Операции и ограничители, составленные из специальных литер:

+	сложение, объединение множеств
-	вычитание, разность множеств
*	умножение, пересечение множеств
/	деление, симметрическая разность множеств
:=	присваивание

```
&      логическое И
~      логическое НЕ
=      равно
# <>   не равно
<      меньше чем
>      больше чем
<=     меньше или равно
>=     больше или равно
( )    круглые скобки
[ ]    индексные скобки
{ }    скобки множества
(* *)  скобки комментария
^      операция разыменования
, . ; : .. |  знаки пунктуации
```

Последовательные лексемы принято разделять одним или несколькими пробелами. Однако необходимо это только в тех случаях, когда отсутствие пробелов привело бы к слиянию двух лексем в одну. Например, во фрагменте "IF x = y THEN" пробелы нужны перед x и после y, а вокруг знака равенства они могут быть опущены.

5. Комментарии могут быть вставлены между любыми двумя лексемами. Они являются произвольными последовательностями литер, заключенными в скобки для комментариев (* и *). Комментарии служат дополнительной информацией для человека и пропускаются компилятором. Они могут также служить для задания режимов работы компилятора.

5. ОПЕРАТОРЫ И ВЫРАЖЕНИЯ

Языковая конструкция, задающая некоторое действие, называется оператором. Операторы могут истолковываться (исполняться), и это истолкование (исполнение) влечет за собой последствия, заключающиеся в том, что изменяется состояние вычислительного процесса, который задается совокупным значением всех переменных программы. Самое элементарное действие — присваивание значения переменной. Присваивание имеет вид

```
$     Присваивание = Обозначение ":=" Выражение.
```

и соответствующее ему действие состоит из трех частей, выполняемых в такой последовательности:

1. Вычислить обозначение, определяющее некоторую переменную.
2. Вычислить выражение, получив некоторое значение.
3. Заменить значение переменной из п. 1 на значение выражения из п. 2.

Простые примеры присваиваний

```
i := 1
x := y + z
```

Здесь i получает значение 1, x — значение суммы y и z, прежние значения теряются. Заметьте, что следующие пары операторов, выполняемые последовательно, дают разные результаты:

```
i := i + 1; j := 2*i
j := 2*i; i := i + 1
```

Полагая начальное значение i равным 0, для первой пары получим i = 1, j = 2, в то время как вторая пара дает j = 0. Если мы захотим обменять значения переменных i и j, то последовательность операторов

```
i := j; j := i
```

не даст желаемого результата. Мы должны ввести вспомогательную переменную, скажем k, для сохранения значения и задать три последовательных присваивания

```
k := i; i := j; j := k
```

В общем случае выражение состоит из операндов и знаков операций. Его вычисление состоит из применения к операндам операций в предписанном порядке, как правило, слева направо. Операндами могут быть константы, переменные или функции. (Функции будут описаны далее.) Вообще говоря, идентификация переменной требует в свою очередь вычисления обозначения; здесь мы, однако, ограничимся лишь случаем использования простой переменной, изображаемой идентификатором. Арифметические выражения (существуют и другие выражения) включают числа, числовые переменные и арифметические операции. К последним относятся основные арифметические операции: сложение (+), вычитание (-), умножение (*) и деление. Все они будут подробно рассмотрены в разделе, посвященном основным типам данных. Здесь же достаточно упомянуть, что знак (/) зарезервирован для деления действительных чисел, а для целых мы используем в качестве знака операции ключевое слово DIV, что означает взятие целой части частного.

Выражение состоит из последовательных слагаемых. Запись

```
T0 + T1 + ... + Tn
```

эквивалентна

```
((T0 + T1) + ...) + Tn
```

Синтаксис выражения определяется правилами

```
$      ПростоеВыражение = ["+"|"-"] Слагаемое
$              {ОперацияТипаСложения Слагаемое}.
$      ОперацияТипаСложения = "+"|"-"|"OR".
```

ПРИМЕЧАНИЕ: Пока читатель может считать, что синтаксические понятия Выражение и ПростоеВыражение эквивалентны. Различие между ними и смысл операций **OR**, **AND** и **NOT** будут разъяснены в разделе, посвященном данным типа **BOOLEAN**.

Аналогичным образом, каждое слагаемое состоит из <u>множителей</u>. Слагаемое

F0 * F1 * ... * Fn

эквивалентно

((F0 * F1) * ...) * Fn

и определяется синтаксически по правилам:

```
$      Слагаемое = Множитель
$              {ОперацияТипаУмножения Множитель}.
$      ОперацияТипаУмножения = "*"|"/"|"DIV"|"MOD"|"AND"|"&".
```

Каждый множитель — это или константа, или переменная, или функция, или выражение, заключенное в круглые скобки.

Примеры арифметических выражений:

```
2 * 3 + 4 * 5          = (2*3)+(4*5)      = 26
15 DIV 4 * 4           = (15 DIV 4)*4     = 12
15 DIV (4 * 4)         = 0
2 + 3 * 4 - 5          = 2+(3*4)-5        = 9
6.25 / 1.25 + 1.5      = 5.0 + 1.5        = 6.5
```

Учитывая, что множитель в свою очередь тоже может быть выражением, очевидно, что синтаксис множителей рекурсивен.

```
$      Множитель = Число | Цепочка | Множество |
$                 Обозначение[ФактическиеПараметры] |
$                 "(" Выражение ")" | "NOT" Множитель.
```

Правила вычисления выражений в действительности очень просты: сложные ситуации встречаются весьма редко, но мы тем не менее укажем несколько основных правил, заслуживающих упоминания.

1. Каждой переменной в выражении должно быть предварительно присвоено значение.
2. Два знака операций не могут стоять рядом. Например, запись **a*-b** неправильна: нужно писать **a*(-b)**.
3. При умножении нельзя пропускать знак операции. Например, запись **2n** неправильна: должно быть **2*n**.
4. ОперацияТипаУмножения имеет более высокий приоритет, чем ОперацияТипаСложения.
5. При возникновении сомнений в правилах вычисления (т.е. старшинстве операций) используйте дополнительные скобки для уточнения. Например, **a+b*c** можно записать и как **a+(b*c)**.

Присваивание — лишь одна из возможных форм операторов. Другие формы будут введены в следующих разделах. Мы перечислим эти формы в виде синтаксического определения

```
$      Оператор = [ Присваивание | ВызовПроцедуры |
$              ЦиклПока | ЦиклДо | ЦиклСШагом |
$              БезусловныйЦикл | УсловныйОператор |
$              ОператорВыбора | ОператорПрисоединения |
$              ОператорВозврата | "EXIT" ].
```

Некоторые из этих форм — структурированные операторы, т.е. их компоненты в свою очередь могут быть операторами. Таким образом, определение операторов, как и выражений, рекурсивно.

Самая фундаментальная структура языка — последовательность. Вычисление — последовательность действий, где каждое действие задается некоторым оператором и исполняется после завершения предшествующего действия. Строгая временная упорядоченность — существенная предпосылка последовательного программирования. Если оператор S1 следует за S0, то мы указываем эту последовательную во времени связь точкой с запятой

S0; S1

Этот <u>разделитель операторов</u> (не завершитель) указывает на то, что за действием, соответствующим S0, должно непосредственно следовать действие, соответствующее S1. Последовательность операторов синтаксически определяется так:

```
$      ПослОператоров = Оператор {";" Оператор}.
```

Синтаксис операторов подразумевает, что оператор может вообще не содержать литер. В таком случае оператор называют пустым, и, очевидно, он задает пустое действие. Эта диковинка среди операторов имеет определенный смысл. Пустой оператор позволяет вставлять точку с запятой в такие места, где она на самом деле избыточна, например в конец последовательности операторов.

6. УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Главная особенность ЭВМ – способность выполнять отдельные действия циклически либо выбирать одно из нескольких действий в зависимости от ранее вычисленных результатов. Таким образом, последовательность выполняемых действий не всегда совпадает с последовательностью соответствующих операторов. Последовательность действий определяется управляющими структурами, указывающими повторение, выбор либо условное выполнение заданных операторов.

6.1. Операторы повторения (циклы)

Наиболее общая ситуация – повторение одного оператора или последовательности под управлением некоторого условия. Повторение продолжается, пока условие остается истинным. Это выражается оператором цикла с условием продолжения (ЦиклПока). Его синтаксис –

```
$       ЦиклПока = "WHILE" Выражение
$                  "DO" ПослОператоров "END".
```

Соответствующее ему действие –

1. Вычислить условие, которое принимает форму выражения со значением или **TRUE** (истина) или **FALSE** (ложь).
2. Если получилось значение **TRUE**, выполнить последовательность операторов, а затем повторить шаг 1; если значение условия – **FALSE**, то закончить выполнение.

Условное выражение в операторе цикла имеет тип **BOOLEAN** (булев, логический). Этот тип будет обсуждаться в разделе, посвященном типам данных. Здесь же достаточно знать, что простое сравнение – выражение типа **BOOLEAN**. Пример цикла был дан во вводном примере, где повторение заканчивалось, когда сравниваемые переменные принимали одинаковые значения. Вот еще примеры операторов цикла с условием продолжения.

1. Пусть вначале q = 0 и r = x; вычислить, сколько раз можно вычесть y из x, т.е. вычислить частное q = x **DIV** y и остаток r = x **MOD** y, если x и y – натуральные числа.

```
WHILE r >= y DO
 r := r - y; q := q + 1
END
```

2. Пусть z = 1 и i = k, умножить z на x k раз, т.е. вычислить z = x^k, если z и k – натуральные числа.


```
WHILE i > 0 DO
 z := z*x; i := i -
END
```

Используя циклы, важно помнить следующее:

1. При каждом повторении должно происходить приближение к цели, т.е. к условию окончания. Очевидным следствием этого является необходимость того, чтобы повторяющиеся вычисления влияли каким-либо образом на условие. Следующие условия либо неверны, либо зависят от некоторых предварительных условий, которые должны выполняться до начала выполнения цикла.

```
WHILE i > 0 DO
 k := 2*k  (* i не изменяется *)
END

WHILE i # 0 DO
 i := i - 2  (* i должно быть четным
                и положительным *)
END

WHILE n # i DO
 n := n*i; i := i + 1
END
```

2. Если условие не выполнено в самом начале, то цикл эквивалентен пустому оператору, т.е. не производит никаких действий.

3. Для того чтобы выяснить, каков эффект выполнения цикла, нужно установить соотношение, сохраняющееся при повторениях и называемое <u>инвариантом</u>. В приведенном выше примере деления инвариант – это уравнение q*y + r = x, выполняющееся перед началом каждого повторения. В примере возведения в степень инвариант – z*x^i = x^k, который вместе с условием i = 0 дает желаемый результат z = x^k.

4. Следует избегать повторения идентичных вычислений (хотя терпение ЭВМ безгранично и она не будет жаловаться). Простое правило – избегать внутри повторяющихся операторов выражений, в которых ни одна переменная не меняет своего значения. Например, оператор

```
WHILE i < 3*N DO
 tab[i] := x + y*z + z*i;
 i := i + 1
END
```

следует записать более эффективно как

```
n := 3*N; u := x + y*z;
WHILE i < n DO
 tab[i] := u + z*i; i := i + 1
END
```

Кроме оператора цикла с условием продолжения имеется оператор цикла с условием окончания (ЦиклДо). Этот цикл выполняется до тех пор, пока условие не станет истинным.

```
$     ЦиклДо = "REPEAT" ПослОператоров "UNTIL" Выражение.
```

Существенное отличие его от первого заключается в том, что условие окончания проверяется всякий раз после (а не до) выполнения последовательности операторов. В результате последовательность всегда выполняется по крайней мере один раз. Преимущество состоит в том, что условие может содержать переменные, значение которых не определено до выполнения цикла.

```
REPEAT
 i := i + 50; j := j + 2; k := i DIV j
UNTIL k > 23

REPEAT
 r := r - y; q := q + 1
UNTIL r < y
```

Два приведенных типа операторов цикла – наиболее распространенные и простые конструкции повторения. Но существуют и другие, в особенности оператор цикла с шагом, который будет описан позже в соответствующем месте. Безусловный цикл – обобщение циклов с условием окончания и условием продолжения, поскольку он позволяет задавать условие окончания в различных местах повторяющейся последовательности операторов. Его завершение осуществляется оператором, состоящим из одного ключевого слова EXIT (выход). Хотя безусловный цикл и удобен в некоторых случаях, мы рекомендуем использовать операторы с пред- и постусловием, поскольку они более ясно выделяют единственное условие завершения в синтаксически очевидной точке.

```
$     БезусловныйЦикл = "LOOP" ПослОператоров "END".
```

6.2. Условные операторы

Условный оператор имеет вид

```
$     УсловныйОператор = "IF" Выражение
$           "THEN" ПослОператоров
$          {"ELSIF" Выражение "THEN" ПослОператоров}
$          ["ELSE" ПослОператоров] "END".
```


Следующий пример иллюстрирует общий вид условного оператора.

```
IF R1 THEN S1
ELSIF R2 THEN S2
ELSIF R3 THEN S3
ELSE S4
END
```

Его смысл очевиден из значения слов (IF – если; THEN – то; ELSE – иначе; ELSIF – иначе, если ...). Следует, однако, помнить, что выражения R1 ... R3 вычисляются одно за другим и что, как только одно из них даст значение TRUE, будет выполнена соответствующая ему последовательность операторов, после чего условный оператор считается завершенным. Последующие условия при этом не проверяются. Примеры:

```
IF x = 0 THEN s := 0
ELSIF x < 0 THEN s := -1
ELSE s := 1
END

IF ODD(k) THEN z := z*x END

IF k > 10 THEN k := k - 10; d := 1
ELSE d := 0
END
```

Рассмотренные нами структуры уже позволяют разработать несколько простых завершенных программ, описанных далее. Первый пример – расширение нашего вводного примера, вычисляющего наибольший общий делитель (нод) двух натуральных чисел x и y. Расширение состоит во введении переменных u и v и операторов, которые позволяют вычислить наименьшее общее кратное (нок) для x и y. Величины нок и нод связаны соотношением

```
нок(x,y)*нод(x,y) = x*y
```

```
MODULE нoднок;
 FROM InOut IMPORT ReadCard,WriteLn,
                   WriteString,WriteCard;
 VAR x,y,u,v: CARDINAL;
BEGIN
 WriteString("x="); ReadCard(x); WriteLn;
 WriteString("y="); ReadCard(y);
 u := x; v := y;
 WHILE x # y DO
  (* нод(x,y) = нод(x0,y0), x*v + y*u = 2*x0*y0 *)
  IF x > y THEN
   x := x - y; u := u + v
```

```
    ELSE
     y := y - x; v := v + u
    END
  END;
  WriteCard(x,6); WriteCard((u + v) DIV 2),6); WriteLn
END ноднок.
```

Это еще один пример вложенных управляющих структур. Повторение, выраженное оператором с предусловием, включает в себя условную структуру, выраженную условным оператором IF, который в свою очередь включает две последовательности операторов, состоящих каждая из двух присваиваний. Эта иерархическая структура выделена соответствующими сдвигами "внутренних" частей.

Другой пример, демонстрирующий иерархическую структуру, вычисляет i-ю степень действительного (REAL) числа x, где i — натуральное число.

```
MODULE Степень;
 FROM InOut IMPORT ReadCard,WriteString,WriteLn;
 FROM RealInOut IMPORT ReadReal,Done,WriteReal;

 VAR i: CARDINAL; x,z: REAL;
BEGIN
 WriteString("x="); ReadReal(x);
 WHILE Done DO
  WriteString("^i="); ReadCard(i);
  z := 1.0;
  WHILE i > 0 DO
   (* z * x^i = x0^i0 *)
   z := z*x; i := i - 1
  END;
  WriteReal(z,16); WriteLn;
  WriteString("x="); ReadReal(x)
 END;
 WriteLn
END Степень.
```

Здесь операторы, вычисляющие степень, охвачены еще одной конструкцией повторения: каждый раз после получения результата запрашивается новая пара x и i. Внешнее повторение управляется логической переменной Done, указывающей, действительно ли введено число x. (Эта переменная импортируется и ее значение устанавливается процедурой чтения ReadReal).

Лобовое вычисление степени многократными умножениями — операция вполне корректная, но не очень экономная. Мы теперь дадим более сложное и более эффективное решение. Оно базируется на следующих соображениях: цель повторения — достигнуть значения i = 0. Это получается последовательным уменьшением i при сохранении инварианта z*x^i = x0^i0, где x0 и i0 обозначают

начальные значения x и i. Более быстрый алгоритм должен, следовательно, основываться на уменьшении i несколько большими шагами. Приведенное здесь решение делит i пополам. Но это возможно, только если i четно. Следовательно, если i нечетно, его нужно уменьшить на 1. Конечно, каждое изменение i должно сопровождаться коррекцией z с целью сохранения инварианта. Отметим одну деталь: уменьшение i на 1 не выражается явно, а осуществляется последующим делением на 2. Еще отметим, что функция ODD(i) (нечетный) равна TRUE, если i — нечетное число, и равна FALSE в противном случае. Идентификаторы x и z обозначают действительные значения в отличие от целых значений. Следовательно, они могут представлять и дроби.

```
MODULE Степень;
 FROM InOut IMPORT ReadCard,WriteString,WriteLn;
 FROM RealInOut IMPORT ReadReal,Done,WriteReal;

 VAR i: CARDINAL; x,z: REAL;
BEGIN
 WriteString("x="); ReadReal(x);
 WHILE Done DO
  WriteString("^i="); ReadCard(i);
  z := 1.0;
  WHILE i > 0 DO
   (* z * x^i = x0^i0 *)
   IF ODD(i) THEN z := z*x END;
   x := x*x; i := i DIV 2
  END;
  WriteReal(z,16); WriteLn;
  WriteString("x="); ReadReal(x)
 END;
 WriteLn
END Степень.
```

Следующий пример программы имеет структуру, почти совпадающую с предыдущей программой. В этом примере вычисляется логарифм по основанию 2 вещественного числа x, значение которого лежит между 1 и 2. Инвариант совместно с условием завершения (b = 0) определяет желаемый результат сумма = log2(x).

```
MODULE Log2;
 FROM InOut IMPORT WriteString,WriteLn;
 FROM RealInOut IMPORT ReadReal,Done,WriteReal;

 VAR x,a,b,сумма: REAL;
BEGIN
 WriteString("x="); ReadReal(x);
 WHILE Done DO
  (* 1.0 <= x < 2.0 *)
```

```
  WriteReal(x,15);
  a := x; b := 1.0; сумма := 0.0;
  REPEAT
   (* log2(x) = сумма + b*log2(a) *)
   a := a*a; b := 0.5*b;
   IF a >= 2.0 THEN
    сумма := сумма + b; a := 0.5*a
   END
  UNTIL b < 1.0E-7;
  WriteReal(сумма,16); WriteLn;
  WriteString("x="); ReadReal(x)
 END;
 WriteIn
END Log2.
```

Обычно процедуры вычисления стандартных математических функций не требуют детального программирования, поскольку они могут быть получены из набора программ, аналогичного модулям ввода и вывода. Такой набор не совсем удачно называется <u>библиотекой программ</u>. В следующем примере, опять демонстрирующем использование оператора REPEAT, применим для вычисления косинуса и показательной функции (exp) подпрограммы из библиотеки, называемой MathLib0. Мы получим таблицу значений для затухающих колебаний. Обычно набор стандартных процедур включает функции sin, cos, exp, ln (натуральный логарифм), sqrt (квадратный корень) и arctan (арктангенс).

```
MODULE Колебания;
 FROM InOut IMPORT ReadCard,WriteString,WriteLn;
 FROM RealInOut IMPORT ReadReal,WriteReal;
 FROM MathLib0 IMPORT exp,cos;

 CONST dx = 0.19634953; (*pi/16*)

 VAR i,n: CARDINAL;
  x,y,r: REAL;

BEGIN
 WriteString("n="); ReadCard(n);
 WriteString("r="); ReadReal(r); WriteLn;
 i := 0; x := 0.0;
 REPEAT x := x + dx; i := i + 1;
  y := exp(-r*x)*cos(x);
  WriteReal(x,15); WriteReal(y,15); WriteLn
 UNTIL i >= n
END Колебания.
```


7. ЭЛЕМЕНТАРНЫЕ ТИПЫ ДАННЫХ

Мы раньше уже говорили о том, что все переменные должны быть описаны. Это означает, что их имена указываются в заголовке программы. Кроме введения имени (что дает компилятору возможность обнаружить и отметить неправильно написанные идентификаторы) описания также связывают с каждой переменной <u>тип данных</u>. Этот тип данных представляет собой статическую информацию о переменной, в отличие, например, от ее значения. Эта информация тоже может способствовать обнаружению в ошибочной программе таких несоответствий, которые могут быть обнаружены простым просмотром программы без ее выполнения.

Тип переменной определяет множество ее возможных значений и операций, применимых к ней. Каждая переменная имеет единственный тип, который можно узнать из ее описания. Каждая операция требует операндов определенного типа и выдает результат тоже определенного типа. Следовательно, из текста программы видно, применима ли данная операция к данной переменной.

В программе можно объявлять новые типы данных. Такие сконструированные типы обычно образуются как композиция основных типов. Существует некоторое количество наиболее часто используемых элементарных типов, называемых <u>стандартными типами</u>, которые являются основными в языке и не требуют описания. О них будет рассказано в этом разделе, хотя некоторые из них уже возникали в предшествующих примерах.

Фактически тип имеют не только переменные, но и константы, функции, операнды и операции (их результаты). В случае констант тип обычно выводим по записи самой константы, другими словами, из ее явного описания.

Сначала мы расскажем о стандартных типах Модулы, а затем рассмотрим форму описаний переменных и констант. Другие типы данных и описаний будут изложены в последующих разделах.

7.1. Тип INTEGER (целый)

Этот тип представляет целые числа, и любому значению типа INTEGER соответствует некоторое целое число. Операции, применимые к типу INTEGER, включают основные арифметические операции

+	сложить
−	вычесть
*	умножить
DIV	разделить
MOD	остаток от деления

Деление нацело, обозначаемое ключевым словом DIV, дает целую часть частного от деления первого операнда на второй.

```
15 DIV 4 = 3
-15 DIV 4 = -3
15 DIV (-4) = -3
```

(*Судя по данному примеру, то, как автор понимает целую часть отрицательного числа (операция truncate), не согласуется с описанием этой операции в книге Д.Кнута "Искусство программирования для ЭВМ" (М.:Мир,1976,т.1,с.68) – "наибольшее целое, меньшее или равное x". – Прим. перев.*) Операция MOD обозначает остаток целочисленного деления. Если мы определим

q = x DIV y, r = x MOD y,

то будет выполняться соотношение

x = q*y + r, 0 <= r < y

Значение x MOD y определено только для положительных x и y.

Изменение знака обозначается унарной операцией – знаком минус. Кроме этой операции существуют еще две унарные операции ABS(x) и ODD(x). Первая дает абсолютное значение величины x, а вторая выдает результат типа BOOLEAN со значением "x нечетно".

На каждой ЭВМ множество значений типа INTEGER ограничено некоторым конечным множеством целых, обычно интервалом -2^(N-1) ... 2^(N-1)-1, где N – небольшое целое число, часто 16 или 32, в зависимости от числа битов, используемых в ЭВМ для представления целых чисел. Если арифметическая операция выдает результат, лежащий за пределами этого диапазона, то говорят, что возникло <u>переполнение</u>. Вычислительная машина прореагирует на это событие соответствующим образом, обычно прекращением процесса вычислений. Программист должен добиваться, чтобы при выполнении его программы переполнения не возникали.

7.2. Тип CARDINAL (натуральный)

Подобно типу INTEGER, тип CARDINAL представляет целые числа, но только неотрицательные значения, т.е. натуральные числа и 0. К этому типу применимы те же операции, что и к INTEGER.

Тип CARDINAL по сравнению с INTEGER имеет то преимущество, что при его использовании явно выражается факт неотрицательности переменной. Мы рекомендуем тип CARDINAL во всех случаях, когда отрицательные значения не будут (или не должны!) возникать....

Явное исключение отрицательных значений из типа CARDINAL требует более тщательного программирования. Например, программист, привыкший к типу INTEGER, вероятно,попадет в такую ловушку. Допустим, что оператор S должен выполняться в цикле с переменной i, принимающей значения N-1,N-2,...,1,0. Операторы


```
i := N - 1;
WHILE i >= 0 DO
 S; i := i - 1
END
```

будут работать правильно, если i имеет тип INTEGER. Но в случае типа CARDINAL произойдет ошибка, поскольку получится -1. Фактически выражение i >= 0 для i типа CARDINAL всегда истинно. Вот правильная форма этого фрагмента программы:

```
i := N;
WHILE i > 0 DO
 i := i - 1; S
END
```

Еще одно преимущество использования типа CARDINAL состоит в том, что вычислительная машина, использующая N битов для представления целых, обеспечит для типа CARDINAL диапазон 0 ... 2^N-1, а максимальное значение INTEGER будет 2^(N-1)-1. Более того, умножение и деление обычно несколько быстрее выполняются над операндами типа CARDINAL.

Модула не разрешает использовать операнды типа INTEGER и CARDINAL в одном и том же выражении (так называемые <u>смешанные выражения</u>). Причина кроется в том, что машинные команды, реализующие операции, различны для этих двух типов. Ситуация несколько облегчается присутствием так называемых <u>функций приведения</u>. Если i имеет тип INTEGER, а c – тип CARDINAL, то выражение i + c запрещено, i + INTEGER(c) будет иметь тип INTEGER, а CARDINAL(i) + c будет типа CARDINAL.

7.3. Тип REAL (действительный)

Значения типа REAL – действительные числа. Имеющиеся операции – это опять же основные арифметические операции и функция ABS. Деление обозначается символом / (вместо DIV).

Константы типа REAL характеризуются тем, что имеют десятичную точку и, возможно, десятичный <u>порядок</u>. Вот примеры таких констант:

1.5 1.50 1.5E2 2.34E-2 0.0

Порядок состоит из прописной латинской буквы E, за которой следует целое. Это означает, что предшествующее действительное число должно быть умножено на 10 в степени "порядок". Следовательно,

1.5E2 = 150.0, 2.34E-2 = 0.0234

Важно помнить, что действительные числа представляются в машине как пары, состоящие из дробной части и порядка. Это называется представлением с плавающей точкой. Конечно, обе части представления содержат конечное число цифр. Вследствие этого представление действительных чисел принципиально неточно, и в вычислениях, использующих такие значения, возникают погрешности, поскольку при выполнении операций может происходить округление или потеря разрядов.

Следующая программа делает очевидным наличие погрешностей, присущих операциям с типом REAL. Эта программа вычисляет гармонический ряд

H(n) = 1 + 1/2 + 1/3 + ... + 1/n

двумя различными способами: один раз суммирование происходит слева направо, а другой раз – справа налево. Согласно законам арифметики, эти две суммы должны быть равны. Однако если происходит потеря цифр (или даже округление), то суммы при больших **n** будут существенно различаться. Правильный способ, очевидно, тот, который начинает с меньших слагаемых.

```
MODULE ГармРяд:
  FROM InOut IMPORT ReadCard,Done,Write,
                    WriteLn,WriteString;
  FROM RealInOut IMPORT WriteReal;

  VAR i,n: CARDINAL;
    x,d,s1,s2: REAL;
BEGIN
  WriteString("n="); ReadCard(n);
  WHILE Done DO
    s1 := 0.0; d := 0.0; i := 0;
    REPEAT
      d := d + 1.0; i := i + 1; s1 := s1 + 1.0/d;
    UNTIL i >= n;
    WriteReal(s1,16); s2 := 0.0;
    REPEAT
      s2 := s2 + 1.0/d; d := d - 1.0; i := i - 1;
    UNTIL i = 0;
    WriteReal(s2,16); WriteLn;
    WriteString("n="); ReadCard(n)
  END;
  WriteLn
END ГармРяд.
```

Главная причина явного разделения действительных и целых чисел – их различное внутреннее представление. Значит, и арифметические операции для разных типов реализуются разными командами. Модула, следовательно, запрещает выражения со

смешанными операндами.

Можно, однако, преобразовать целые числа в действительные (более точно: внутреннее представление целых может быть преобразовано в представление с плавающей точкой) и обратно явными функциями преобразования, а именно

FLOAT(c) TRUNC(x)

Функция FLOAT(c) имеет тип REAL и представляет значение величины c типа CARDINAL; TRUNC(x) представляет целую часть действительной величины x и имеет тип CARDINAL. Программист должен иметь в виду, что различные реализации Модулы могут предоставлять другие либо дополнительные функции преобразования.

7.4. Тип BOOLEAN (логический)

Значение типа BOOLEAN (булев, логический) имеет два логических, истинностных значения, обозначаемых стандартными идентификаторами TRUE и FALSE. Булевы переменные обычно обозначаются идентификаторами, имеющими смысл прилагательных, причем значение TRUE подразумевает наличие соответствующего свойства, а FALSE – его отсутствие. Имеется набор логических операций, которые вместе с переменными типа BOOLEAN образуют выражения этого типа. Логическими операциями являются AND (и) (обозначаемая также &), OR (или) и NOT (не) (обозначаемая также ~). Их смысл таков:

p AND q = "как p, так и q равны TRUE"
p OR q = "или p, или q, или оба равны TRUE"
NOT p = "p равно FALSE"

Точное определение операций, однако, немного другое, хотя результат тот же:

p AND q = IF p THEN q ELSE FALSE
p OR q = IF p THEN TRUE ELSE q

Эти определения подразумевают, что второй операнд может и не вычисляться, если результат уже известен после вычисления первого операнда. Замечательное свойство этого определения заключается в том, что результат выражения может иметь смысл, даже если второй операнд не определен. Как следствие этого порядок операторов может оказаться существенным.

Иногда можно упростить логическое выражение применением простых правил преобразования. Особенно полезны законы де Моргана, задающие эквивалентность

(NOT p) AND (NOT q) = NOT (p OR q)
(NOT p) OR (NOT q) = NOT (p AND q)

Сравнения выдают результат типа **BOOLEAN**, т.е. **TRUE**, если сравнение справедливо, и **FALSE** – если нет. Например,

```
7 = 12        FALSE
7 < 12        TRUE
1.5 >= 1.6    FALSE
```

Такие сравнения синтаксически классифицируются как выражения, а два сравниваемых операнда – это простые выражения (см. раздел, посвященный выражениям и операторам). Результат сравнения имеет тип **BOOLEAN** и может быть использован в управляющих структурах, таких, как условный оператор или операторы цикла. Знак # означает "не равно" (его синоним : <>).

```
$      Выражение = ПростоеВыражение
$                  [Сравнение ПростоеВыражение].
$      Сравнение = "="|"#"|"<>"|"<"|"<="|">"|">="|"IN".
```

Следует отметить, что, подобно арифметическим операциям, среди логических операций тоже имеется отношение старшинства. **NOT** имеет наивысший приоритет, затем следует **AND** (называемая также логическим умножением), а затем **OR** (логическое сложение) и, наконец, операции сравнения. Как и в случае арифметических выражений, можно свободно применять скобки, чтобы явно выразить связь между операциями. Вот примеры логических выражений:

```
x = y
(x <= y) & (y < z)
(x > y) OR (y >= z)
NOT p OR q
```

Заметим, что такая конструкция, как x < y **AND** z < w, запрещена.

Значения типа **BOOLEAN** можно сравнивать, причем не только на равенство. В частности,

```
FALSE < TRUE
```

Следовательно, логическая импликация "из p следует q" выражается или как

```
(NOT p) OR q,    или как    p <= q.
```

Приведенный пример обращает внимание на правило, гласящее, что операнды операции (включая сравнения) должны иметь одинаковый тип. Значит, следующие сравнения неверны:

```
1 = TRUE
5 = 5.0
i + j = p OR q
```


Неверна также, например, запись: x <= y < z, которую можно развернуть в (x <= y) **AND** (y < z). Следующие логические выражения тем не менее верны:

```
i + j < k - m
p OR q = (i < j)
```

И последнее замечание: хотя запись p = **TRUE** и верна, но так писать считается плохим стилем, лучше – просто p. Аналогично вместо p = **FALSE** лучше писать **NOT** p (или ~p).

7.5. Тип CHAR (литерный)

Любая вычислительная система осуществляет связь с окружающим ее миром через некоторые устройства ввода и вывода. Они читают или печатают элементы, взятые из некоторого фиксированного множества литер. Это множество образует диапазон значений типа **CHAR**. К сожалению, различные типы вычислительных машин могут использовать различные множества литер, что делает связь между машинами (т.е. обмен программами и данными) трудной и запутанной. Существует, однако, международный стандартизированный набор литер, так называемый набор ISO. Стандарт ISO определяет набор из 128 литер, причем 33 из них – так называемые управляющие литеры. Оставшиеся 95 – видимые печатаемые (или графические) литеры, показанные в следующей таблице. Набор литер упорядочен, и каждая литера имеет фиксированное порядковое число. Например, латинская буква A – это 66-я литера: она имеет порядковое число 65. Стандарт ISO оставляет, однако, в таблице несколько мест незаполненными, их можно заполнять различными литерами в соответствии с национальными стандартами. Наиболее широко применяется американский стандарт, называемый **ASCII** (**American Standart Code for Information Interchange** – американский стандартный код для обмена информацией). Здесь мы приводим именно набор литер **ASCII**. Порядковое число литеры получается сложением чисел, соответствующих столбцу и строке, которые содержат нужную литеру. Эти числа обычно приводятся в восьмеричном виде, и мы тоже последуем этому правилу. Первые два столбца содержат управляющие литеры, они обычно обозначаются сокращениями, указывающими на их предполагаемую функцию. Их смысл, однако, не заложен в код, а определяется только их интерпретацией конкретным устройством. Поэтому достаточно просто помнить, что эти литеры обычно не печатаются.

Константы типа **CHAR** обозначаются литерой, заключенной в кавычки или апострофы. Литерные значения могут быть только присвоены переменным типа **CHAR**, но не могут использоваться в арифметических операциях. Арифметические операции можно применять лишь к порядковым числам, получаемым с помощью функции преобразования ORD(ch). И наоборот, литера, имеющая порядковый

номер n, может быть получена функцией преобразования CHR(n). Эти две взаимно обратные функции связаны уравнениями

```
CHR(ORD(ch)) = ch   и   ORD(CHR(n)) = n
```

которые верны при 0 <= n < 128. Они позволяют получить числовое значение цифры ch как

```
ORD(ch) - ORD("0")
```

и вычислить цифру, представляющую числовое значение n, как

```
CHR(n + ORD("0"))
```

Эти две формулы используют сплошное расположение цифр в стандарте ISO, причем ORD("0") = 60B = 48. Они, как правило, применяются в подпрограммах преобразования последовательностей цифр в числа и наоборот, чисел в последовательности цифр. Следующий фрагмент программы читает цифры и присваивает значение числа, представленного ими в десятичной форме, переменной x.

```
x := 0; Read(ch);
WHILE ("0" <= ch) & (ch <= "9") DO
  x := 10*x + (ORD(ch) - ORD("0")); Read(ch)
END
```

Таблица литер кода ASCII

	0	20	40	60	100	120	140	160
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
10	bs	can	(	8	H	X	h	x
11	ht	em	)	9	I	Y	i	y
12	lf	sub	*	:	J	Z	j	z
13	vt	esc	+	;	K	[	k	{
14	ff	fs	,	<	L	\	l	|
15	cr	gs	-	=	M	]	m	}
16	so	rs	.	>	N	^	n	~
17	si	us	/	?	O	_	o	del

Управляющие литеры используются для различных целей, в основном для управления функционированием внешних устройств, но, кроме того, для разбиения текста на строки и его структуризации. Важная функция управляющих литер – указывать конец строки или конец страницы текста. Нет никакого общепринятого стандарта на их использование в этих целях. Мы обозначим управляющую литеру, отмечающую конец строки текста, идентификатором EOL (End Of Line – конец строки); ее конкретное значение зависит от используемой вычислительной системы.

Для представления невидимых символов Модула использует их порядковое число в восьмеричном представлении, за которым следует буква C (латинская). Например, 14C – значение типа CHAR, обозначающее управляющую литеру ff (перевод формата), имеющую порядковое число 14B.

7.6. Тип BITSET

Величины, принадлежащие типу BITSET, – множества целых чисел между 0 и N-1, где N – константа, определяемая конкретной вычислительной машиной. Это обычно либо длина машинного слова, либо небольшое кратное ей число. Константы этого типа обозначаются как множества (см. также раздел, посвященный типу множество). Вот примеры:

```
{5,7,11}  {0}  {8..15}  {0..3,11,15}  {}
```

Обозначение m .. n – сокращение для m,m+1, ... ,n-1,n.

```
$    Множество = [КвалИдент]
                 "{" [Элемент {"," Элемент}] "}".
$    Элемент = Выражение [".." Выражение].
```

Над множествами определены операции:

- \+ объединение множеств
- \- разность множеств
- * пересечение множеств
- / симметрическая разность множеств

Считая, что i обозначает элемент множества, а u, v – множества, эти операции можно определить в терминах принадлежности к множеству следующими равенствами:

```
i IN (u + v) = (i IN u) OR (i IN v)
i IN (u - v) = (i IN u) AND NOT (i IN v)
i IN (u * v) = (i IN u) AND (i IN v)
i IN (u / v) = ((i IN u) # (i IN v))
```

Операция проверки принадлежности элемента множеству считается

операцией сравнения. Выражение i IN u имеет тип BOOLEAN. Оно принимает значение TRUE, если i — элемент множества u. Тип BITSET представляется в машине как множество битов, т.е. характеристической функцией множества. Например, i-й бит в u равен 1, если i принадлежит u, и равен 0 в противном случае. Следовательно, множественные операции реализованы как логические операции над N членами множественной переменной. Поэтому такие операции очень эффективны, и время их выполнения обычно даже меньше, чем при сложении целых чисел.

8. ОПИСАНИЯ КОНСТАНТ И ПЕРЕМЕННЫХ

Уже упоминалось, что все идентификаторы, используемые в программе, должны быть описаны в ее заголовке (или импортированы из некоторого другого модуля). Это не касается стандартных идентификаторов, известных во всех программах.

Если идентификатор должен обозначать константное значение, то его можно ввести описанием константы, которое указывает, какую величину будет заменять этот идентификатор. Описание константы имеет вид

```
$       ОписаниеКонстанты = Идентификатор "=" КонстВыражение.
$       КонстВыражение = Выражение.
```

КонстВыражение — выражение, содержащее только константы. Более точно, оно должно быть вычислимо без выполнения программы, простым просмотром ее текста. Последовательности описаний констант предшествует ключевое слово CONST. Пример:

```
CONST N = 16;
 EOL = 36C;
 пусто = {};
 M = N - 1;
```

Константы с явными именами облегчают чтение и разбор программ лишь в том случае, когда им даны подходящие имена. Если, например, идентификатор N используется вместо значения константы по всей программе, то эту константу можно изменить, исправив текст программы только в одном месте, а именно в описании N. Это предотвращает распространенную ошибку, когда при изменении константы некоторые ее вхождения остаются незамеченными и сохраняют старое значение, что ведет к несоответствиям в программе.

Описание переменной похоже на описание константы. Вместо значения константы ставится тип переменной, который в некотором смысле может считаться константным свойством переменной. Вместо знака равенства используется двоеточие.


```
$       ОписаниеПеременной = СписИдент ":" Тип.
$       СписИдент = Идентификатор {"," Идентификатор}.
```

В одном описании могут быть перечислены несколько переменных одного и того же типа. Последовательности описаний переменных предшествует слово VAR. Пример:

```
VAR i,j,k: CARDINAL;
 x,y,z: REAL;
 ch: CHAR
```

9. МАССИВЫ

До сих пор мы давали каждой переменной отдельное имя. Однако это неудобно, когда много переменных одного типа нужно обрабатывать одинаковым образом, как, например, при создании некоторой таблицы данных. В этом случае хотелось бы дать всему множеству переменных единственное имя и обозначать отдельный элемент идентифицирующим его номером, так называемым индексом. Такой тип данных называют структурированным (более точно: структурированный тип массив). В следующем примере переменная a состоит из N элементов, имеющих тип CARDINAL, а индексы меняются от 0 до N-1.

```
VAR a: ARRAY [1..N-1] OF CARDINAL
```

Элемент в этом случае обозначается идентификатором массива, за которым следует выбирающий индекс, например a[i], где i — выражение, значение которого должно лежать внутри диапазона изменения индекса, заданного в описании массива. Синтаксически, a[i] — обозначение, а выражение i — индексное выражение. Если, например, всем элементам массива a нужно присвоить нулевое значение, то это удобно выразить оператором цикла, в котором индекс при каждом повторении получает новое значение.

```
i := 0;
REPEAT a[i] := 0; i := i + 1
UNTIL i = N
```

Этот пример иллюстрирует ситуацию, возникающую настолько часто, что Модула предлагает специальную управляющую конструкцию, выражающую то же самое более лаконично. Она называется оператором цикла с шагом:

```
FOR i := 0 TO N-1 DO
 a[i] := 0
END
```

Общая форма этого оператора такова:

```
$       ЦиклСШагом = "FOR" Идентификатор ":=" Выражение
$                    "TO" Выражение ["BY" КонстВыражение]
$                    "DO" ПослОператоров "END".
```

Выражения, стоящие до и после ключевого слова TO, определяют соответственно начальную и конечную границы диапазона изменения так называемой <u>управляющей переменной</u> или <u>параметра цикла</u> (i). Необязательная конструкция, начинающаяся с ключевого слова BY, определяет шаг увеличения (или уменьшения, если он отрицателен) параметра цикла. По умолчанию значение шага принимается равным единице.

Рекомендуется использовать цикл с шагом только в простых случаях; в частности, операнды выражений, определяющих диапазон, и в особенности сам параметр цикла, не должны меняться повторяемыми операторами. Значение параметра цикла после завершения цикла следует считать неопределенным.

Дополнительные примеры призваны продемонстрировать использование массивов и операторов цикла с шагом. В первом примере вычисляется сумма N элементов массива.

```
сумма := 0;
FOR i := 0 TO N-1 DO
 сумма := a[i] + сумма
END
```

Во втором примере требуется найти минимальный элемент в массиве и его индекс. Инвариантом цикла является условие min = минимум(a[0],...,a[i-1]).

```
min := a[0]; k := 0;
FOR i := 1 TO N-1 DO
 IF a[i] < min THEN
  k := i; min := a[k]
 END
END
```

В третьем примере алгоритма <u>сортировки</u> массива в возрастающем порядке мы используем второй пример как фрагмент.

```
FOR i := 0 TO N-2 DO
 min := a[i]; k := i;
 FOR j := i TO N-1 DO
  IF a[j] < min THEN
   k := j; min := a[k]
  END
 END;
 a[k] := a[i]; a[i] := min
END
```


Пусть нужно скопировать массив a в массив b. Можно попробовать записать это так:

```
FOR i := 0 TO N-1 DO
 b[i] := a[i]
END
```

Хотя это и правильно, однако то же самое выражается проще оператором a := b. Отсюда видно, что оператор присваивания можно *применять и непосредственно к массивам.*

Очевидно, что цикл с шагом хорошо соответствует случаям обработки <u>всех</u> элементов внутри заданного диапазона. Но если, например, мы хотим найти индекс элемента, равного заданной *величине x, то нам неизвестно заранее, сколько элементов массива* придется просмотреть. Следовательно, здесь рекомендуется использовать цикл с условием продолжения или с условием окончания. Этот алгоритм называется <u>линейным поиском</u>.

```
i := 0;
WHILE (i < N) & (a[i] # x) DO
 i := i + 1
END
```

Взяв отрицание условия продолжения и применяя закон де Моргана, мы получим, что при завершении цикла будет удовлетворяться условие (i = N) OR (a[i] = x). Если второе подвыражение истинно, то это значит, что искомый элемент найден, а i — его индекс; если же i = N, то все a[i] не равны x.

Обратим внимание на то, что условие завершения составное. Известен стандартный способ его упрощения. Вспомним, что повторение *должно завершиться, если либо найден нужный элемент,* либо достигнут конец массива. Хитрость состоит в пометке конца массива специальным элементом, равным x, на котором поиск автоматически прекратится. Для этого нужно всего лишь добавить в конец массива вспомогательный элемент a[N], который будет служить ограничителем.

```
a: ARRAY [0..N] OF CARDINAL;

a[N] := x; i := 0;
WHILE a[i] # x DO i := i + 1 END
```

Если после завершения приведенного фрагмента программы i = N, то ни один исходный элемент не равен x, если же i не равно N, то i — нужный индекс.

Более сложная задача — поиск элемента, равного x, в упорядоченном массиве, т.е. при условии, что a[i-1] <= a[i] для всех i = 1...N-1. Лучший метод — это так называемый <u>двоичный поиск</u>: проверить средний элемент массива, а затем применить этот

же метод к левой или правой половине. Этот алгоритм реализует следующий далее фрагмент программы (предполагается, что N > 0). Инвариантами цикла являются условия (a[k] < x для всех k = 0...i-1) и (a[k] > x для всех k = j+1...N-1)

```
i := 0; j := N-1; найден := FALSE;
REPEAT центр := (i + j) DIV 2;
 IF x < a[центр] THEN j := центр - 1
 ELSIF x > a[центр] THEN i := центр + 1
 ELSE найден := TRUE
 END
UNTIL (i > j) OR найден
```

Поскольку каждый шаг делит интервал поиска пополам, то число необходимых сравнений равно всего лишь log2(N). Приведем другую, более эффективную версию, исключающую составное условие завершения.

```
i := 0; j := N-1;
REPEAT центр := (i + j) DIV 2;
 IF x <= a[центр] THEN j := центр - 1 END;
 IF x >= a[центр] THEN i := центр + 1 END
UNTIL i > j;
IF i > j+1 THEN найден ELSE не найден END
```

Ниже приведена еще более изощренная версия. Ее основная идея – не прекращать поиск сразу, как только элемент найден, поскольку это довольно редкое событие по сравнению с числом безуспешных проверок.

```
i := 0; j := N-1;
REPEAT центр := (i + j) DIV 2;
 IF x < a[центр] THEN j := центр
 ELSE i := центр + 1
 END
UNTIL i >= j
```

На этом завершим список примеров использования простых массивов.

Все элементы массива имеют одинаковый тип, однако сам элемент может быть в свою очередь массивом (фактически, как мы увидим далее, это может быть любой структурированный тип). Массив массивов называется <u>многомерным массивом</u> или <u>матрицей</u>, поскольку можно считать, что каждый индекс соответствует одному измерению в декартовом пространстве. Вот примеры двумерных массивов:

```
a: ARRAY [1..N],[1..N] OF REAL
T: ARRAY [0..M-1],[0..N-1] OF CHAR
```

представляющие собой сокращение следующих полных форм:


```
a: ARRAY [1..N] OF
     ARRAY [1..N] OF
       REAL

T: ARRAY [0..M-1] OF
     ARRAY [0..N-1] OF
       CHAR
```

Отступы используются, чтобы отразить иерархическую структуру описания. Общий синтаксис типа массив таков:

```
$ ТипМассив = "ARRAY" ПростойТип {"," ПростойТип}
$             "OF" Тип.
```

Здесь простой тип, обозначающий диапазон индексов, имеет либо вид

```
"[" КонстВыражение ".." КонстВыражение "]"
```

либо является идентификатором. Например, описание массива

```
map: ARRAY[" ".."~"] OF CARDINAL
```

вводит массив из 95 чисел типа CARDINAL, причем каждый элемент индексируется графической литерой, как это показано в следующих операторах:

```
map["A"] := 0; k := map["+"]
```

Синтаксис обозначений допускает сокращение, аналогичное сокращениям в описаниях: можно писать a[i,j] вместо a[i][j]. Однако последняя форма отчетливее выражает тот факт, что [j] является индексом массива a[i]. Синтаксис обозначения элемента массива:

```
$ Обозначение = КвалИдент {"[" СписВыражений "]"}.
$ СписВыражений = Выражение {"," Выражение}.
```

При работе с матрицами оператор цикла с шагом демонстрирует все свои преимущества, особенно в числовых применениях. Канонический пример – перемножение двух матриц, где каждый элемент произведения c = a*b определяется как

```
c[i,j] = a[i,1]*b[1,j] + a[i,2]*b[2,j] + ...
                              ... + a[i,N]*b[N,j]
```

Пусть имеются описания

```
a: ARRAY [1..M],[1..K] OF REAL;
b: ARRAY [1..K],[1..N] OF REAL;
c: ARRAY [1..M],[1..N] OF REAL
```

Алгоритм умножения состоит из трех вложенных друг в друга циклов

```
FOR i := 1 TO M DO
 FOR j := 1 TO N DO
  сумма := 0.0;
  FOR k := 1 TO K DO
   сумма := a[i,k]*b[k,j] + сумма
  END;
  c[i,j] := сумма
 END
END
```

Во втором примере мы демонстрируем поиск слова в таблице. Каждое слово в таблице – массив литер. Предполагаем, что таблица T описана так же, как в одном из приведенных выше примеров, а x задан следующим образом:

```
x: ARRAY [0..N-1] OF CHAR
```

Наше решение использует типичный линейный поиск:

```
i := 0; найден := FALSE;
WHILE ~найден&(i < M) DO
 найден := "T[i] равен x";
 i := i + 1
END
```

Если определить равенство двух слов x и y как x[j] = y[j] для всех j = 0...N-1, то можно выразить "внутренний" поиск в таком виде:

```
j := 0; равно := TRUE;
WHILE равно&(j < N) DO
 равно := T[i,j] = x[j]; j := j + 1
END;
найден := равно
```

Это решение выглядит довольно неуклюжим, однако в случае M > 0 и N > 0 его можно преобразовать в более простую форму. Окончательный вариант алгоритма поиска в таблице можно записать так:

```
i := 0;
REPEAT j := 0;
 REPEAT B := T[i,j] # x[j]; j := j + 1
  UNTIL B OR (j = N);
  i := i + 1
 UNTIL NOT B OR (i = M)
```


Значение логической переменной B в конце фрагмента имеет следующий смысл: "слово x не найдено".

Мы провели достаточную подготовительную работу для разработки содержательных завершенных программ. Здесь будет представлено три примера полностью завершенных программ, причем все они содержат массивы.

В первом примере задача состоит в печати списка степеней двойки, причем каждая строка должна содержать величины 2^i, i, 2^(-i). Эта задача была бы очень простой, если использовать тип REAL. Тогда ядро программы выглядело бы так:

```
d := 1; f := 1.0;
FOR exp := 1 TO N DO
 d := 2*d; печать(d);   (* d = 2^exp *)
 печать(exp);
 f := f/2.0; печать(f)  (* f = 2^(-exp) *)
END
```

Однако мы хотим получить точные результаты с тем количеством цифр, которое потребуется. По этой причине мы представляем и целое число d = 2^exp, и дробь f = 2^(-exp) массивами "цифр", каждая из которых лежит в диапазоне 0...9. Для представления f нам потребуется N цифр, для d – только логарифм от N. Отметим, что удвоение d производится справа налево, а деление f пополам – слева направо. Таблица результатов приведена ниже.

```
MODULE СтепениДвойки;
 FROM InOut IMPORT Write,WriteLn,
                   WriteString,WriteCard;
 CONST M = 11; N = 32; (* M ~ N*log(2) *)
 VAR i,j,k,exp: CARDINAL;
  c,r,t: CARDINAL;
  d: ARRAY [0..M] OF CARDINAL;
  f: ARRAY [0..N] OF CARDINAL;
BEGIN
 d[0] := 1; k := 1;
 FOR exp := 1 TO N DO
  (* вычислить d = 2^exp операциями d := 2*d *)
  c := 0; (* перенос *)
  FOR i := 0 TO k-1 DO
   t := 2*d[i] + c;
   IF t >= 10 THEN
    d[i] := t - 10; c := 1
   ELSE
    d[i] := t; c := 0
```

```
         2   1 0.5
         4   2 0.25
         8   3 0.125
        16   4 0.0625
        32   5 0.03125
        64   6 0.015625
       128   7 0.0078125
       256   8 0.00390625
       512   9 0.001953125
      1024  10 0.0009765625
      2048  11 0.00048828125
      4096  12 0.000244140625
      8192  13 0.0001220703125
     16384  14 0.00006103515625
     32768  15 0.000030517578125
     65536  16 0.0000152587890625
    131072  17 0.00000762939453125
    262144  18 0.000003814697265625
    524288  19 0.0000019073486328125
   1048576  20 0.00000095367431640625
   2097152  21 0.000000476837158203125
   4194304  22 0.0000002384185791015625
   8388608  23 0.00000011920928955078125
  16777216  24 0.000000059604644775390625
  33554432  25 0.0000000298023223876953125
  67108864  26 0.00000001490116119384765625
 134217728  27 0.000000007450580596923828125
 268435456  28 0.0000000037252902984619140625
 536870912  29 0.00000000186264514923095703125
1073741824  30 0.000000000931322574615478515625
2147483648  31 0.0000000004656612873077392578125
4294967296  32 0.00000000023283064365386962890625
```

Вывод программы СтепениДвойки

```
 END
END;
IF c > 0 THEN
 d[k] := 1; k := k + 1
END;
(* вывод d[k-1]...d[0] *) i := M;
REPEAT i := i - 1; Write(" ") UNTIL i = k;
REPEAT i := i - 1; Write(CHR(d[i]+ORD("0")))
UNTIL i = 0;
WriteCard(exp,4);
(* вычисление f = 2^(-exp) операциями f := f DIV 2
   и его вывод *)
WriteString(" 0."); r := 0; (* остаток *)
```



```
  FOR j := 1 TO exp-1 DO
   r := 10*r + f[j]; f[j] := r DIV 2;
   r := r MOD 2; Write(CHR(f[j]+ORD("0")))
  END;
  f[exp] := 5; Write("5"); WriteLn
 END
END СтепениДвойки.
```

Наш второй пример имеет аналогичную природу. Задача – точно вычислить десятичные дроби $d = 1/i$. Трудность заключается, конечно, в представлении дробей, которые являются бесконечными последовательностями цифр (например, 1/3 = 0.333...). К счастью, все дроби имеют повторяющийся период, и было бы разумно и полезно отмечать начало периода и завершать вычисление в его конце. Но как находить начало и конец периода? Рассмотрим сначала алгоритм вычисления цифр дроби.

Начиная с остатка ост = 10, мы повторяем умножение его на 10 и делим произведение на i. Частное от деления нацело – это новое значение для ост. Этот алгоритм в точности воспроизводит стандартный метод деления, что иллюстрируется следующим фрагментом программы и числовым примером при i = 7.

```
1.000000 / 7 = 0.142857
1 0
  30
   20
    60
     40
      50
       1

ост := 1;
REPEAT ост := 10*ост;
 СледЦифра := ост DIV i;
 ост := ост MOD i
UNTIL ...
```

Мы знаем, что период завершится, как только получится остаток, который встречался ранее. Поэтому наш способ – запоминать остатки и их индексы. Индексы обозначают то место, откуда начинается период. Обозначим массив индексов через x и дадим его элементам начальные значения 0. Индексация массива ведется по значениям остатков. В рассмотренном выше примере деления на 7 величины индексов следующие: x[1]=1, x[2]=3, x[3]=2, x[4]=5, x[5]=6, x[6]=4.

```
MODULE Дроби;
 FROM InOut IMPORT Write,WriteLn,
                   WriteString,WriteCard;
 CONST Основание = 10; N = 32;
```

```
 2 0.5'0
 3 0.'3
 4 0.25'0
 5 0.2'0
 6 0.1'6
 7 0.'142857
 8 0.125'0
 9 0.'1
10 0.1'0
11 0.'09
12 0.08'3
13 0.'076923
14 0.0'714285
15 0.0'6
16 0.0625'0
17 0.'0588235294117647
18 0.0'5
19 0.'052631578947368421
20 0.05'0
21 0.'047619
22 0.0'45
23 0.'0434782608695652173913
24 0.041'6
25 0.04'0
26 0.0'384615
27 0.'037
28 0.03'571428
29 0.'0344827586206896551724137931
30 0.0'3
31 0.'032258064516129
32 0.03125'0
```

Вывод программы Дроби.

```
 VAR i,j,m: CARDINAL;
  ост: CARDINAL;
  d: ARRAY [1..N] OF CARDINAL; (* цифры *)
  x: ARRAY [0..N] OF CARDINAL; (* индекс *)
BEGIN
 FOR i := 2 TO N DO
  FOR j := 0 TO i-1 DO x[j] := 0 END;
  m := 0; ост := 1;
  REPEAT m := m + 1; x[ост] := m;
   ост := Основание * ост; d[m] := ост DIV i;
   ост := ост MOD i
  UNTIL x[ост] # 0;
  WriteCard(i,6); WriteString(" 0.");
  FOR j := 1 TO x[ост]-1 DO Write(CHR(d[j]+ORD("0"))) END;
```



```
  Write("'");
  FOR j := x[ост] TO m DO Write(CHR(d[j]+ORD("0"))) END;
  WriteLn
 END
END Дроби.
```

В последнем примере рассмотрим программу печати списка простых чисел. Она основывается на проверке делимости последовательных целых чисел. Проверяемые целые числа получаются увеличением предыдущего числа по очереди на 2 и на 4, тем самым сразу отсекаются числа, кратные 2 и 3. Необходимо проверять делимость только на простые числа, которые были уже ранее вычислены и запомнены.

```
MODULE ПростыеЧисла;
 FROM InOut IMPORT WriteLn,WriteCard;

 CONST N = 250; M = 16; (* M^2 ~ N *)
  LL = 8; (* сколько простых чисел печатать на строке *)

 VAR i,k,x: CARDINAL;
  шаг,граница,квадрат,L: CARDINAL;
  простое: BOOLEAN;
  P,V: ARRAY [0..M] OF CARDINAL;
BEGIN L := 0;
 x := 1; шаг := 4; граница := 1; квадрат := 9;
 FOR i := 3 TO N DO
  (* найти следующее простое число P[i] *)
  REPEAT x := x + шаг; шаг := 6 - шаг;
   IF квадрат <= x THEN
    граница := граница + 1; V[граница] := квадрат;
    квадрат := P[граница+1]*P[граница+1]
   END;
   k := 2; простое := TRUE;
   WHILE простое & (k < граница) DO
    k := k + 1;
    IF V[k] < x THEN
     V[k] := V[k] + 2*P[k]
    END
    простое := x # V[k]
   END
  UNTIL простое;
  IF i <= M THEN P[i] := x END;
  WriteCard(x,6); L := L + 1;
  IF L = LL THEN
   WriteLn; L := 0
  END
 END
END ПростыеЧисла.
```

5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61
67	71	73	79	83	89	97	101
103	107	109	113	127	131	137	139
149	151	157	163	167	173	179	181
191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271
277	281	283	293	307	311	313	317
331	337	347	349	353	359	367	373
379	383	389	397	401	409	419	421
431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521
523	541	547	557	563	569	571	577
587	593	599	601	607	613	617	619
631	641	643	647	653	659	661	673
677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787
797	809	811	821	823	827	829	839
853	857	859	863	877	881	883	887
907	911	919	929	937	941	947	953
967	971	977	983	991	997	1009	1013
1019	1021	1031	1033	1039	1049	1051	1061
1063	1069	1087	1091	1093	1097	1103	1109
1117	1123	1129	1151	1153	1163	1171	1181
1187	1193	1201	1213	1217	1223	1229	1231
1237	1249	1259	1277	1279	1283	1289	1291
1297	1301	1303	1307	1319	1321	1327	1361
1367	1373	1381	1399	1409	1423	1427	1429
1433	1439	1447	1451	1453	1459	1471	1481
1483	1487	1489	1493	1499	1511	1523	1531
1543	1549	1553	1559	1567	1571	1579	1583
1597	1601						

Вывод программы ПростыеЧисла.

Эти примеры завершают первую часть книги. Они показывают, что массивы являются фундаментальным средством, используемым в большинстве программ. Едва ли существует хоть одна программа, написанная с практической, а не учебной целью, которая бы не использовала циклов и массивов (или аналогичных им структур данных).

Часть 2

10. ПРОЦЕДУРЫ

Рассмотрим задачу обработки некоторого набора данных, состоящего из заголовка и последовательности из N отдельных подобных элементов. В общем виде это можно записать так:

```
ВводЗаголовка;
ОбработкаЗаголовка;
ПечатьЗаголовка;
FOR i := 1 TO N DO
 ВводЭлемента;
 ОбработкаЭлемента;
 Печать(i); ПечатьЭлемента
END
```

Описание исходной задачи дано в терминах подзадач, причем выделена лишь основная структура задачи, а детали опущены. Конечно, подзадачи ВводЗаголовка, ОбработкаЗаголовка и т.д. должны быть далее описаны со всеми необходимыми деталями. Однако можно не заменять слова-описатели соответствующими фрагментами программ на Модуле, а рассматривать эти слова как идентификаторы и определить детали подзадач в текстуально отделенных частях программы, называемых процедурами (или подпрограммами). Такие определения называются описаниями процедур. Они называются так потому, что определяют действия процедуры и дают ей имя. Идентификаторы в главной программе, соответствующие описаниям процедур, называются вызовами процедур и их действие заключается в выполнении соответствующей процедуры. С точки зрения синтаксиса, вызов процедуры представляет собой оператор.

Процедуры играют фундаментальную роль при разработке программ. Они помогают выявлять структуру алгоритма и разбивать программу на логически связанные единицы. Это особенно важно в случае сложных алгоритмов, т.е. длинных программ. Хотя применение в приведенном выше примере отдельно описанных процедур, а не прямой подстановки текста на место идентификаторов может показаться довольно расточительным, тем не менее ради получения ясной структуры программы часто рекомендуется использовать явные процедуры даже в таком простом

случае. Процедура, разумеется, особенно полезна, если она должна вызываться из различных мест программы.

Описание процедуры состоит из ключевого слова PROCEDURE и следующего за ним идентификатора (вместе они образуют заголовок процедуры), далее, ключевого слова BEGIN и последовательности операторов, называемой телом процедуры. В тексте программы тело процедуры заменено ее идентификатором. Описание заканчивается ключевым словом END и повторением идентификатора процедуры. Это позволяет компилятору выявлять отсутствие завершителей операторов и описаний. Общий синтаксис описания процедуры будет дан позже. Приведем простой пример, в котором вычисляется сумма a[0]+...+a[N-1].

```
PROCEDURE Сложить;
BEGIN сумма := 0.0;
 FOR i := 0 TO N-1 DO
  сумма := a[i] + сумма
 END
END Сложить
```

Концепция процедуры становится гораздо полезнее благодаря двум ее дополнительным особенностям. Это параметры и свойство локальности имен. Параметры делают возможным при вызове одной и той же процедуры из различных мест программы применять эту процедуру к различным величинам и переменным, задаваемым в месте вызова. Понятие локальности имен и объектов значительно повышает роль процедур в структуризации программы и выделении ее частей. Далее в первую очередь мы будем обсуждать понятие локальности. Подводя итог всему сказанному, повторим следующие существенные моменты:

1. Процедура помогает выявить внутреннюю структуру программы и облегчает декомпозицию программистской задачи на подзадачи.
2. Если процедура вызывается из двух или более мест, то ее применение сокращает программу, а следовательно, уменьшает вероятность программистских ошибок. Дополнительным преимуществом является уменьшение размера откомпилированного кода.

11. ПОНЯТИЕ ЛОКАЛЬНОСТИ

Если мы посмотрим на пример процедуры из предыдущей главы, то заметим, что роль переменной i строго ограничена телом процедуры. Это свойство локальности следует выразить явно, что можно сделать, описав i внутри процедуры. Тем самым i становится локальной переменной.

```
PROCEDURE Сложить;
 VAR i: CARDINAL;
BEGIN сумма := 0.0;
 FOR i := 0 TO N-1 DO
  сумма := a[i] + сумма
 END
END Сложить
```


В некотором смысле описание процедуры принимает вид отдельной программы. Фактически любые описания, допустимые в программе (констант, типов, переменных или процедур), могут появиться и в описании процедуры. Отсюда следует, что описания процедур могут быть вложенными и что их определение рекурсивно.

Считается хорошей привычкой описывать локальные объекты, т.е. ограничивать область существования объекта той процедурой, в которой он имеет смысл. Процедура, т.е. фрагмент текста программы, для которого действительно описание имени, называется его областью видимости. Так как описания могут быть вложенными, то вложимы друг в друга и области видимости. Возможность существования объектов, локальных в некоторой области видимости, влечет за собой некоторые интересные следствия. Можно, например, использовать одно и то же имя для обозначения разных объектов. Фактически это самое полезное следствие, поскольку программист свободен в выборе идентификаторов и не должен знать, какие идентификаторы существуют в окружающей области видимости, если они не обозначают объектов, используемых в локальной области (в противном случае программист обязан их знать). Такое разделение информации о различных частях программы особенно полезно, а возможно, даже жизненно необходимо в случае больших программ.

Правила для областей видимости таковы:

1. Областью видимости идентификатора является процедура, в которой он описан, и все процедуры, заключенные в ней, кроме тех, которые подчиняются правилу 2.
2. Если идентификатор i, описанный в процедуре P, повторно описан в некоторой внутренней процедуре Q, заключенной в P, то процедура Q и все заключенные в ней процедуры исключаются из области видимости идентификатора i, описанного в P.
3. Стандартные идентификаторы Модулы считаются описанными в воображаемой процедуре, охватывающей всю программу.

Эти правила можно запомнить с помощью алгоритма поиска описания данного идентификатора i: вначале просматриваются описания процедуры P, в теле которой встретился i; если среди них не встретилось описание для i, то продолжается поиск в процедуре, охватывающей P; далее повторяется это же правило, пока не встретится нужное описание.

```
VAR a: CARDINAL;
PROCEDURE P;
 VAR b: CARDINAL;
```

```
PROCEDURE Q;
 VAR b,c: BOOLEAN;
BEGIN
 (* a,b(BOOLEAN),c  видимы *)
END Q;
BEGIN
 (* a,b(CARDINAL)  видимы *)
END P
```

Следствием концепции локальности и правила о том, что переменная не существует за пределами ее области видимости, является тот факт, что значение переменной теряется, когда описывающая ее процедура завершается. Предполагается, что при повторном вызове процедуры значение такой переменной неизвестно. Величины локальных переменных не определены при входе в процедуру (первом или повторном). Следовательно, если переменная должна сохранять значение между двумя вызовами, ее нужно описывать за пределами процедуры. "Время жизни" переменной – это время, в течение которого активна описавшая ее процедура.

Использование локальных описаний имеет три существенных преимущества:

1. Становится ясным, что объект заключен в процедуре, которая обычно составляет лишь малую часть всей программы.
2. Есть гарантия, что случайное использование локального объекта другими частями программы будет обнаружено компилятором.
3. Возможна минимизация используемой памяти при реализации, поскольку память для переменных освобождается при завершении процедуры, в которой эти переменные локальны. Память затем может быть повторно использована для других переменных.

12. ПАРАМЕТРЫ

Процедуры могут иметь параметры. Именно эта существенная деталь делает процедуры такими полезными. Рассмотрим еще раз пример процедуры Сложить. Весьма вероятно, что в программе содержится несколько массивов, к которым можно было бы ее применить. Переписывать ее для каждого из них – громоздко и некрасиво. Этого можно избежать, введя операнд в качестве параметра процедуры.

```
PROCEDURE Сложить(VAR x: Вектор);
 VAR i: CARDINAL;
BEGIN сумма := 0.0;
 FOR i := 0 TO N-1 DO
  сумма := x[i] + сумма
 END
END Сложить
```


В список параметров заголовка процедуры введен параметр x. Тем самым он автоматически становится локальным объектом, подменяющим фактический массив, указанный в вызове процедуры.

```
Сложить(a);...;Сложить(b)
```

Массивы a и b называются фактическими параметрами, замещающими массив x, называемый формальным параметром. При задании формального параметра должен указываться его тип. Это позволяет компилятору проверить, подходит ли тип подставляемого фактического параметра. Мы говорим, что фактические параметры a и b должны быть совместимы с формальным параметром x. В приведенном выше примере его тип – Вектор, который предположительно описан в окружении процедуры Сложить таким образом:

```
TYPE Вектор = ARRAY [0..N-1] OF REAL;

VAR a,b: Вектор
```

Еще лучшая версия процедуры может включать в качестве параметров не только массив, но и результат суммирования. Мы позже еще вернемся к этому примеру, но прежде нужно объяснить, что существует два вида формальных параметров: параметр-переменная и параметр-значение. Первый отмечается ключевым словом VAR, второй – его отсутствием.

В завершение этого раздела дадим синтаксис описания процедуры и вызова процедуры.

```
$      ОписаниеПроцедуры = ЗаголовокПроцедуры ";"
$                          Блок Идентификатор.
$      ЗаголовокПроцедуры = "PROCEDURE" Идентификатор
$                          [ФормальныеПараметры].
$      Блок = {Описание} ["BEGIN" ПослОператоров] "END".
$      ФормальныеПараметры = "(" [ФПСекция {";" ФПСекция}] ")"
$                          [":" КвалИдент].
$      ФПСекция = ["VAR"] СписИдент ":" ФормТип.
$      ФормТип = ["ARRAY" "OF"] КвалИдент.
$      ВызовПроцедуры = Обозначение [ФактическиеПараметры].
$      ФактическиеПараметры = "(" [СписВыражений] ")".
```

Теперь мы знакомы со всеми формами описаний, кроме описания модуля.

```
$      Описание = "CONST" {ОписаниеКонстанты ";"}|
$           "TYPE" {ОписаниеТипа ";"}|
$           "VAR" {ОписаниеПеременной ";"}|
$           ОписаниеПроцедуры ";" | ОписаниеМодуля ";".
```

12.1. Параметры-переменные

Как можно заключить из названия, фактический параметр, соответствующий формальному параметру-переменной (отмеченному словом VAR), должен быть переменной. Идентификатор формального параметра заменяет эту переменную. Пример:

```
PROCEDURE обмен(VAR x,y: CARDINAL);
 VAR z: CARDINAL;
BEGIN z := x; x := y; y := z
END обмен
```

Вызовы процедуры

```
обмен(a,b); обмен(A[i],A[i+1])
```

дают тот же результат, что и выполнение трех записанных ранее присваиваний с учетом соответствующих подстановок при вызове.

Относительно параметров-переменных важно помнить следующее:

1. Параметры-переменные могут служить для передачи из процедуры вычисленного значения.
2. Формальный параметр подменяет подставляемый фактический параметр.
3. Фактический параметр не может быть выражением, а следовательно, и константой, даже если соответствующему формальному параметру и не присваивается значение.
4. Если фактический параметр содержит индексы, то их вычисление происходит в момент подстановки фактических параметров при вызове процедуры.
5. Типы формального и фактического параметров должны совпадать.

12.2. Параметры-значения

Параметры-значения служат для передачи величин из вызывающей среды в процедуру, и они используются в преобладающем большинстве случаев. Соответствующий фактический параметр – выражение, частным (и простейшим) случаем которого являются переменная или константа. Формальный параметр-значение можно рассматривать как локальную переменную указанного типа. При вызове вычисляется фактическое выражение и результат присваивается локальной переменной. Отсюда следует, что формальному параметру далее может быть присвоено новое значение, причем это никак не повлияет на операнды выражения. В известном смысле можно считать, что фактическое выражение и формальный параметр после входа в процедуру разъединяются. В качестве иллюстрации запишем приведенную ранее программу возведения в степень как процедуру.


```
PROCEDURE Степень(VAR z: REAL; x: REAL; i: CARDINAL);
BEGIN z := 1.0;
 WHILE i > 0 DO
  IF ODD(i) THEN z := z*x END;
  x := x*x; i := i DIV 2
 END
END Степень
```

Вот примеры ее допустимых вызовов:

```
Степень(u,2.5,3)
Степень(A[i],B[i],2)
```

При работе с параметрами-значениями следует иметь в виду, что, поскольку формальный параметр представляет локальную переменную, для нее требуется память. Это может оказаться существенным, если тип параметра является массивом из многих элементов. В этом случае рекомендуется задавать параметр-переменную, даже если этот параметр используется только для передачи значения внутрь процедуры.

Обратите внимание на то, что в приведенном примере процедуры параметры z и x объявлены в различных секциях формальных параметров, поскольку запись "VAR z,x: REAL" отнесла бы x тоже к параметрам-переменным, сделав невозможным подстановку вместо него выражения общего вида.

12.3. Гибкие массивы-параметры

Если тип формального параметра является массивом, то соответствующий фактический параметр должен иметь тот же тип. Имеется в виду, что фактический массив должен иметь элементы того же типа и совпадающие границы диапазона индексов. Часто это ограничение оказывается слишком серьезным и поэтому желательна большая гибкость. Это обеспечивается посредством так называемого <u>гибкого массива</u>, который требует только, чтобы типы элементов формального и фактического массивов совпадали, и оставляет свободным диапазон индекса. В этом случае в качестве фактических параметров могут подставляться массивы любого размера (с любым числом элементов). Гибкий массив задается типом элемента, которому предшествует "ARRAY OF". Например, процедура, описанная как

```
PROCEDURE P(s: ARRAY OF CHAR)
```

разрешает вызовы с массивами символов, имеющими любые границы индексов. Нижняя граница диапазона индексов в случае гибкого формального массива равна нулю. Верхняя граница получается посредством вызова стандартной функции HIGH(s). Ее величина

равна числу элементов минус 1. Отсюда следует, что если массив, описанный как

```
a: ARRAY [m..n] OF CHAR
```

подставляется вместо гибкого массива s, то s[i] обозначает a[m+i] для i = 0...HIGH(s), где HIGH(s) = n-m.

13. ПРОЦЕДУРЫ-ФУНКЦИИ

Мы познакомились с двумя возможностями передачи результата из тела процедуры в место вызова: результат присваивается или нелокальной переменной, или параметру-переменной. Существует и третий метод: процедура-функция. Она позволяет использовать вычисленный результат (как непосредственное значение) в выражении. Идентификатор процедуры-функции обозначает как вычисление, так и вычисленный результат. Описание процедуры-функции характеризуется указанием типа результата после списка параметров. В качестве примера преобразуем ранее приведенную процедуру вычисления степени в процедуру-функцию.

```
PROCEDURE степень(x: REAL; i: CARDINAL): REAL;
 VAR z: REAL;
BEGIN z := 1.0;
 WHILE i > 0 DO
  IF ODD(i) THEN z := z*x END;
  x := x*x; i := i DIV 2
 END;
 RETURN z
END степень
```

Возможные вызовы таковы:

```
u := степень(2.5,3)
A[i] := степень(B[i],2)
u := x + степень(y,i+1)/степень(z,i-1)
```

Оператор, передающий результат, состоит из ключевого слова RETURN, за которым следует выражение, вычисляющее результат. <u>Оператор возврата</u> может встретиться в нескольких местах в теле процедуры: он прекращает выполнение тела и осуществляет возврат в место вызова. Обычно, однако, оператор возврата помещается непосредственно перед завершающим END процедуры. Операторы возврата могут также использоваться в обычных процедурах, но в этом случае за словом RETURN не следует выражение. Это средство может служить сигналом аварийного завершения. Подразумевается, что оператор возврата неявно помещается в конце каждой процедуры.

$ ОператорВозврата = "RETURN" [Выражение].

Вызов внутри выражения называется <u>обозначением функции</u>. Его синтаксис такой же, как и у вызова процедуры. Однако список параметров обязателен, хотя он может быть и пустым. Обратимся теперь еще раз к предыдущему примеру суммирования элементов массива и запишем его как процедуру-функцию.

```
PROCEDURE сумма(VAR a: Вектор; n: CARDINAL): REAL;
 VAR i: CARDINAL; s: REAL;
BEGIN s := 0.0;
 FOR i := 0 TO n-1 DO
  s := a[i] + s
 END;
 RETURN s
END сумма
```

Эта процедура в том виде, как она записана, суммирует элементы a[0]...a[n-1], где n задано как параметр-значение и может отличаться от числа элементов N (но не превосходить его!). В более изящном решении a описывается как гибкий массив, поэтому явное указание размера массива отсутствует.

```
PROCEDURE сумма(VAR x: ARRAY OF REAL): REAL;
 VAR i: CARDINAL; s: REAL;
BEGIN s := 0.0;
 FOR i := 0 TO HIGH(x) DO
  s := x[i] + s
 END;
 RETURN s
END сумма
```

Очевидно, что процедура может выдавать более одного результата, осуществляя присваивания различным переменным. Однако лишь одна величина может быть возвращена как результат функции. Более того, ее тип не может быть структурированным. Следовательно, другие результаты должны передаваться в место вызова через параметры вида VAR или через переменные, нелокальные в процедуре-функции. Рассмотрим для примера процедуру, вычисляющую основной результат, определенный как значение функции, и побочный, используемый для подсчета числа вызовов процедуры.

```
PROCEDURE квадрат(x: CARDINAL): CARDINAL;
BEGIN n := n + 1;
 RETURN x*x
END квадрат
```

В этом примере нет ничего примечательного пока побочный

результат используется по прямому назначению (указанному выше). Возможно, однако, и ошибочное использование:

```
m := квадрат(m) + n
```

Здесь побочный результат входит как операнд в выражение, содержащее обозначение самой функции. Следствием этого является, например, тот факт, что значения

```
квадрат(m) + n   и   n + квадрат(m)
```

различаются, явно бросая вызов основному закону коммутативности сложения.

Присваивания значений из процедур-функций нелокальным переменным называются побочными эффектами. Программисту следует отчетливо осознавать их способность приводить к неожиданным результатам в случае их неосторожного использования.

Подведем итог сказанному:

1. Процедура-функция определяет результат, который используется в месте вызова как операнд выражения.
2. Результат процедуры-функции не может быть структурированным.
3. Если процедура-функция выдает вспомогательные результаты, то говорят, что она имеет побочный эффект. Использование таких процедур требует особой аккуратности. Если процедура-функция передает результаты посредством параметров-переменных, то желательно вместо нее использовать обычную процедуру.
4. Рекомендуем выбирать в качестве идентификаторов функций имена существительные. Существительное в этом случае обозначает результат функции. Булевым функциям вполне подходят прилагательные. Обычную же процедуру следует обозначать глаголом, описывающим ее действие.

14. РЕКУРСИЯ

Процедуры могут не только вызываться откуда-то, но и сами себя вызывать. Так как вызвать можно любую видимую процедуру, то процедура может вызвать и себя. Такая самоактивация называется рекурсией. Ее использование уместно, когда алгоритм определен рекурсивно и, в особенности, когда он применяется к рекурсивно определенной структуре данных. Рассмотрим в качестве примера задачу печати всех возможных перестановок n различных объектов. Назвав эту операцию Переставить(n), мы можем записать ее алгоритм следующим образом:

Сначала оставить a[n] на своем месте и сгенерировать все перестановки объектов a[1]...a[n-1], вызвав процедуру

Переставить(n-1), затем повторить процесс, поменяв a[n] местами с a[i] при i = 1, повторить это для всех значений i = 2...n-1. Этот рецепт записывается в виде программы следующим образом (в качестве переставляемых объектов используются литеры):

```
MODULE Перестановка;
 FROM InOut IMPORT Read,Write,WriteLn;

 VAR n: CARDINAL; ch: CHAR;
  a: ARRAY [1..20] OF CHAR;

 PROCEDURE Вывод;
  VAR i: CARDINAL;
 BEGIN
  FOR i := 1 TO n DO Write(a[i]) END;
  WriteLn
 END Вывод;

 PROCEDURE Переставить(k: CARDINAL);
  VAR i: CARDINAL; t: CHAR;
 BEGIN
  IF k = 1 THEN Вывод
  ELSE Переставить(k-1);
   FOR i := 1 TO k-1 DO
    t := a[i]; a[i] := a[k]; a[k] := t;
    Переставить(k-1);
    t := a[i]; a[i] := a[k]; a[k] := t;
   END
  END
 END Переставить;

BEGIN Write(">"); n := 0; Read(ch);
 WHILE ch > " " DO
  n := n + 1; a[n] := ch; Write(ch); Read(ch)
 END;
 WriteLn; Переставить(n)
END Перестановка.
```

Результаты, полученные в случае использования трех литер, таковы:

```
ABC BAC CBA BCA ACB CAB
```

Каждая цепочка рекурсивных вызовов должна на каком-то шаге завершиться, и, следовательно, любая рекурсивная процедура должна содержать рекурсивный вызов внутри условного оператора. В приведенном примере рекурсия завершается, когда число объектов, которые нужно переставить, становится равным единице.

Число возможных перестановок легко вывести из рекурсивного

определения алгоритма. Мы выразим это число как функцию np(n). Пусть задано n элементов, тогда существует n вариантов выбора элемента a[n], и при каждом фиксированном a[n] мы получаем np(n-1) перестановок. Следовательно, общее число np(n) = n*np(n-1). Очевидно, что np(1) = 1. Вычисление величины np можно теперь выразить как вызов рекурсивной процедуры.

```
PROCEDURE np(n: CARDINAL): CARDINAL;
BEGIN
  IF n <= 1 THEN RETURN 1
  ELSE RETURN n*np(n-1)
  END
END np
```

В функции np мы узнаем факториал, который вычисляется как

```
np = 1 * 2 * 3 * ... * n
```

Эта формула наводит на мысль об алгоритме, использующем вместо рекурсии повторение:

```
PROCEDURE np(n: CARDINAL): CARDINAL;
 VAR p: CARDINAL;
BEGIN p := 1;
 WHILE n > 1 DO
  p := n*p; n := n-1
 END;
 RETURN p
END np
```

Этот вариант вычислит результат более эффективно, чем рекурсивная версия. Причина в том, что каждый вызов требует некоторых дополнительных "административных" операций, на выполнение которых тоже расходуется время. Команды, обеспечивающие повторение, тратят меньше времени. И хотя эта разница не может быть слишком уж существенной, рекомендуется все же использовать циклические конструкции вместо рекурсивных всегда, когда это можно сделать без особых усилий. В принципе, это возможно всегда, однако циклическая версия в состоянии настолько усложнить алгоритм и затуманить его смысл, что минусов окажется намного больше, чем плюсов. Например, циклическая форма процедуры перестановки намного сложнее и запутаннее приведенной рекурсивной. Для иллюстрации полезности рекурсии дадим два дополнительных примера. Приводимые далее алгоритмы обычно возникают в задачах, решение в которых легко находится и объясняется с применением рекурсии.

Первый пример принадлежит классу алгоритмов, работающих с данными, структура которых тоже определяется рекурсивно. Характерной задачей такого рода является преобразование простых

выражений в соответствующую постфиксную форму или польскую инверсную запись (Полиз), т.е. в форму, при которой знак операции следует за операндами. Выражение в данном случае будет определяться в виде РБНФ так:

```
Выражение = Слагаемое {("+"|"-") Слагаемое}.
Слагаемое = Множитель {("*"|"/") Множитель}.
Множитель = Буква | "(" Выражение ")"
            | "[" Выражение "]".
```

Обозначая слагаемые как T0,T1, а множители – как F0,F1, запишем правила преобразования:

```
T0 + T1    —>  T0 T1 +
T0 - T1    —>  T0 T1 -
F0 * F1    —>  F0 F1 *
F0 / F1    —>  F0 F1 /
(E)        —>  E
[E]        —>  E
```

Следующая далее программа Полиз вводит выражения с терминала и проверяет входную информацию на синтаксическую правильность. В случае правильного ввода информация в неизменном виде повторяется на выходе, если же ввод неверен, то такой выдачи информации не происходит. Процедура вывода Write импортируется не из стандартного модуля Terminal, а из модуля TextWindows (текстовые окна), управляющего размещением на экране так называемых окон и выдачей на них текста. Мы можем считать, что одно окно используется для повторения принятой входной информации, а второе – для выдачи преобразованных выражений, т.е. основной выходной информации.

Наша программа в точности отражает структуру синтаксиса входных выражений. Поскольку синтаксис рекурсивен, рекурсивна и сама программа. Такое точное отражение – лучшая гарантия правильности программы. Заметим также, что, аналогично, итерация в синтаксисе, выражаемая фигурными скобками, ведет к итерации в программе, выражаемой оператором с условием повторения. В этой программе ни одна из процедур не вызывает себя непосредственно. Здесь рекурсия имеет непрямой характер и возникает, благодаря обращению в процедуре Выражение к Слагаемое, которая в свою очередь обращается к Множитель, а уже Множитель обращается к Выражение. Очевидно, что непрямая рекурсия менее наглядна и заметна, чем прямая.

Этот пример иллюстрирует также применение локальных процедур. Примечателен тот факт, что процедура Множитель описана локальной для процедуры Слагаемое, а та в свою очередь локальна в процедуре Выражение. Это сделано в соответствии с тем правилом, что объекты следует описывать преимущественно локальными в той области, где они используются. Это правило иногда может

оказаться не только желательным, но и необходимым, как иллюстрируется переменными ОпСлож (локальная в Слагаемое) и ОпУмнож (локальная в Множитель). Если бы эти переменные были описаны как глобальные, то программа не смогла бы работать. Для объяснения этого мы должны вспомнить то правило, что локальные переменные существуют (и под них выделяется память) лишь в тот промежуток времени, когда процедура активна. Непосредственным следствием этого в случае рекурсивного вызова является создание нового экземпляра локальной переменной. Таким образом, существует столько экземпляров переменной, сколько уровней рекурсии, из чего, в частности, заключаем, что программист должен заботиться о том, чтобы глубина рекурсии не оказалась чересчур большой.

```
MODULE Полиз;
 FROM Terminal IMPORT Read;
 FROM TextWindows IMPORT
  Window,OpenTextWindow,Write,WriteLn,CloseTextWindow;

 CONST EOL = 36C;
 VAR ch: CHAR; w0,w1: Window;

 PROCEDURE Выражение;
  VAR ОпСлож: CHAR;

  PROCEDURE Слагаемое;
   VAR ОпУмнож: CHAR;

   PROCEDURE Множитель;
   BEGIN
    IF ch = "(" THEN
     Write(w0,ch); Read(ch); Выражение;
     WHILE ch # ")" DO Read(ch) END
    ELSIF ch = "[" THEN
     Write(w0,ch); Read(ch); Выражение;
     WHILE ch # "]" DO Read(ch) END
    ELSE
     WHILE (ch < "a") OR (ch > "z") DO Read(ch) END;
     Write(w1,ch)
    END;
    Write(w0,ch); Read(ch)
   END Множитель;

  BEGIN (*Слагаемое*) Множитель;
   WHILE (ch = "*") OR (ch = "/") DO
    Write(w0,ch); ОпУмнож := ch; Read(ch);
    Множитель; Write(w1,ОпУмнож)
   END
  END Слагаемое;

 BEGIN (*Выражение*) Слагаемое;
  WHILE (ch = "+") OR (ch = "-") DO
   Write(w0,ch); ОпСлож := ch; Read(ch);
   Слагаемое; Write(w1,ОпСлож)
  END
 END Выражение;

BEGIN OpenTextWindow(w0,50,50,300,400,"ввод");
   (*Открыть текстовое окно*)
 OpenTextWindow(w1,400,100,300,400,"вывод");
 Write(w0,">"); Read(ch);
 WHILE ch >= EOL DO
  Выражение;
  WriteLn(w0); WriteLn(w1);
  Write(w0,">"); Read(ch)
 END;
 CloseTextWindow(w1); CloseTextWindow(w0)
END Полиз.
```


Образец данных, обрабатываемых и генерируемых программой, приведен ниже.

```
>a+b              ab+
>a*b+c            ab*c+
>a+b*c            abc*+
>a*(b/[c-d])      abcd-/*
```

Следующий пример программы демонстрирует рекурсию в применении к классу задач поиска решения методом проб и ошибок. В этом методе используется последовательное построение частичных "решений" и проверка их допустимости. Каждое новое частичное решение получается дополнением некоторого другого. Если полученное частичное решение неудовлетворительно, то происходит возврат к частичному решению, из которого оно было получено, и попытка дополнить его другим способом. Поэтому такой подход называется еще поиском с возвратами. Выбор решения задачи осуществляется среди множества частичных решений. Для записи таких алгоритмов очень удобно применение рекурсии.

Наш конкретный пример предназначен для поиска всех возможных размещений 8 ферзей на шахматной доске так, чтобы ни один из них не находился под ударом остальных, т.е. на каждой вертикали, горизонтали и диагонали должно находиться не более одного ферзя. Метод состоит в помещении на j-ю вертикаль (начиная с j = 8) еще одного ферзя; при этом предполагается, что все вертикали справа уже содержат правильно размещенные фигуры. Если на вертикали j нет допустимого места, то нужно пересмотреть размещение ферзя на (j+1)-й вертикали. Информация, необходимая для заключения о том, доступна ли данная клетка, содержится в трех глобальных переменных типа массив: Гориз, Диагон1, Диагон2 так, что

Гориз[i]&Диагон1[i+j]&Диагон2[N+i-j] = "клетка на горизонтали i и вертикали j свободна"

Программа использует набор процедур, импортируемых из модуля, называемого LineDrawing (вычерчивание прямых), чтобы представить выходные данные в легко воспринимаемой графической форме. В частности, вызов процедуры

```
area(c,x,y,w,h) (* область(...) *)
```

закрасит прямоугольник с координатами левого нижнего угла x и y, шириной w и высотой h, "цветом" c. Очевидно, что эта процедура может быть использована как для прочерчивания линий между полями шахматной доски, так и для закрашивания отдельных клеток.

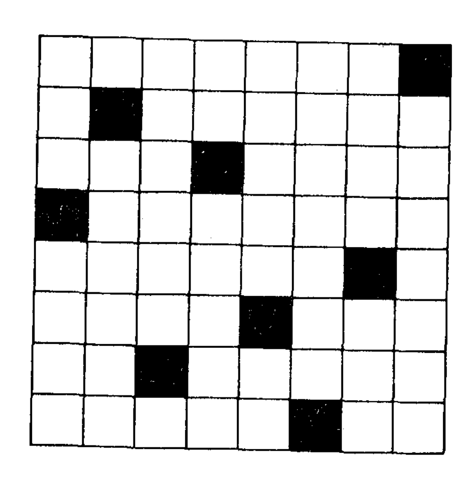
Рекурсия возникает непосредственно в процедуре НоваяВертикаль. Вспомогательные процедуры ПоставитьФерзя и УдалитьФерзя могли бы быть, в принципе, описаны локальными внутри процедуры НоваяВертикаль. Однако, поскольку существует только одна доска (представленная переменными Гориз, Диагон1, Диагон2), эти процедуры соответственно считаются приписанными глобальным данным, а следовательно, не должны быть локальными в (каждой активации) НоваяВертикаль.

```
MODULE Ферзи;
 FROM LineDrawing IMPORT width,height,area,clear;

 CONST N = 8; (* число вертикалей и горизонталей *)
  L = 512; (* размер доски *)
  M = L DIV N; (* размер полей *)

 VAR x0,y0: CARDINAL; (* начальные координаты на доске *)
  Гориз: ARRAY [1..N] OF BOOLEAN;
   (* Гориз[i] = "на i-й горизонтали нет ферзей" *)
  Диагон1: ARRAY [2..2*N] OF BOOLEAN;
   (* Диагон1[i] = "на i-й нисходящей диагонали нет ферзей *)
  Диагон2: ARRAY [1..2*N-1] OF BOOLEAN;
   (* Диагон2[i] = "на i-й восходящей диагонали нет ферзей *)

 PROCEDURE НарисоватьДоску;
  VAR i,j,x,y: CARDINAL;
 BEGIN clear(1);
  FOR i := 1 TO N DO Гориз[i] := TRUE END;
  FOR i := 2 TO 2*N DO Диагон1[i] := TRUE END;
  FOR i := 1 TO 2*N-1 DO Диагон2[i] := TRUE END;
  x0 := (width-L) DIV 2; x := x0;
  y0 := (height-L) DIV 2; y := y0;
  area(3,x0,y0,L,L);
  FOR i := 0 TO N DO
   area(0,x0,y,L,2); y := y + M;
```

Вывод программы Ферзи.

```
   area(0,x,y0,2,L); x := x + M;
  END
 END НарисоватьДоску;

 PROCEDURE Пауза;
  VAR n: CARDINAL;
 BEGIN n := 50000;
  REPEAT n := n-1 UNTIL n = 0
 END пауза;

 PROCEDURE ПоставитьФерзя(i,j: CARDINAL);
 BEGIN
  Гориз[i] := FALSE;
  Диагон1[i+j] := FALSE; Диагон2[N+i-j] := FALSE;
  area(0,x0+2+(j-1)*M,y0+2+(i-j)*M,M-2,M-2)
 END ПоставитьФерзя;

 PROCEDURE УбратьФерзя(i,j: CARDINAL);
 BEGIN
  Гориз[i] := TRUE;
  Диагон1[i+j] := TRUE; Диагон2[N+i-j] := TRUE;
  area(3,x0+2+(j-1)*M,y0+2+(i-1)*M,M-2,M-2)
 END УбратьФерзя;

 PROCEDURE НоваяВертикаль(j: CARDINAL);
  VAR i: CARDINAL;
```

```
BEGIN i := N;
 REPEAT
  IF Гориз[i]&Диагон1[i+j]&Диагон2[N+i-j] THEN
   ПоставитьФерзя(i,j);
   IF j > 1 THEN НоваяВертикаль(j-1)
    ELSE Пауза
   END;
   УбратьФерзя(i,j)
  END;
  i := i-1
 UNTIL i = 0
END НоваяВертикаль;

BEGIN НарисоватьДоску; НоваяВертикаль(N); clear(3) END Ферзи.
```

Часть 3

15. ОПИСАНИЯ ТИПОВ

Каждое описание переменной задает тип этой переменной как ее неизменное свойство. Типом может быть один из стандартных, примитивных типов либо же тип может быть описан в самой программе. Описания типов имеют форму

```
$       ОписаниеТипа = Идентификатор "=" Тип.
```

Им предшествует ключевое слово **TYPE**. Типы делятся на <u>неструктурированные</u> и <u>структурированные</u>. По существу каждый тип определяет множество значений, которые может принимать переменная данного типа. Значение неструктурированного типа – неделимый объект, в то время как величина структурированного типа имеет <u>компоненты</u> (элементы). Например, тип CARDINAL – неструктурированный: его значение неделимо. Не имеет смысла ссылаться, скажем, к третьему биту значения 13; тот факт, что число может "иметь третий бит" или вторую цифру – характеристика его (внутреннего) представления, которое намеренно оставлено скрытым.

В последующих разделах мы опишем способы описания неструктурированных и структурированных типов. Кроме стандартных типов, с которыми мы встречались до сих пор, как неструктурированные типы описываются перечислимый тип и тип диапазона. Из всех структурированных типов мы до сих пор имели дело только с массивами. Кроме этого существует еще тип запись и тип множество. Возможность введения структур, которые динамически изменяются во время исполнения программы, основана на понятии указателей. Эта возможность будет обсуждаться в отдельном разделе.

```
$       Тип = ПростойТип|ТипМассив|ТипЗапись|
$            ТипМножество|ТипУказатель|ТипПроцедура.
$       ПростойТип = КвалИдент|Перечисление|ТипДиапазон.
```

Прежде чем приступить к рассмотрению различных способов задания типов, сделаем одно общее замечание. Если тип T описан так:

```
TYPE T = НекоторыйТип
```

а переменная t:

```
VAR t: T
```

то эти два описания можно слить в одно:

```
VAR t: НекоторыйТип
```

однако в этом случае тип переменной t не будет иметь явного имени.

Понятие типа играет важную роль в программировании, поскольку типы делят все множество переменных в программе на непересекающиеся классы. Ошибочные присваивания между элементами разных классов могут, следовательно, быть обнаружены простым просмотром текста программы без ее выполнения. Пусть имеются такие описания:

```
VAR b: BOOLEAN;
    i: INTEGER;
    c: CARDINAL;
```

в этом случае присваивание b := i невозможно, поскольку типы переменных b и i несовместимы. Два типа называются <u>совместимыми</u>, если они описаны как равные либо удовлетворяют определенным правилам совместимости, которые будут обсуждаться далее. Важным исключением из этих правил являются типы INTEGER и CARDINAL. По этой причине присваивание i := c допустимо.

Для демонстрации правил совместимости рассмотрим следующие описания:

```
TYPE A = ARRAY [0..99] OF CHAR;
     B = ARRAY [0..99] OF CHAR;
     C = A
```

В этом случае переменные типа A можно присваивать переменным типа C (и наоборот), но не переменным типа B. Однако допустимо присваивание b[i] := a[i], поскольку обе переменные имеют один тип CHAR.

16. ПЕРЕЧИСЛИМЫЕ ТИПЫ

Можно описать новый неструктурированный тип <u>перечисление</u> или перечислимый тип, явно выписав все множество значений, которые принадлежат этому типу. Объявление типа

```
T = (c1,c2,...,cn)
```


вводит новый, неструктурированный тип T, для значений которого используются n идентификаторов-констант c1, c2,...,cN. Только эти идентификаторы входят в значения данного типа. Синтаксис описания перечислимого типа следующий:

```
$    Перечисление = "(" СписИдент ")".
```

Операции над величинами такого типа должны описываться программистом как процедуры. Кроме операций присваивания возможны еще сравнения этих величин. Величины упорядочены: c1 – наименьшая, cn – наибольшая. Вот примеры перечислимых типов:

```
TYPE цвет = (красный,оранжевый,желтый,зеленый,
             голубой,синий,фиолетовый);
    ДеньНедели = (пн,вт,ср,чт,пт,сб,вс);
    месяц = (янв,февр,март,апр,май,июнь,июль,авг,
             сент,окт,нояб,дек)
```

Порядковый номер константы ci можно получить, воспользовавшись стандартной функцией ORD(ci): ее значение i-1. Например,

```
ORD(красный) = 0, ORD(чт) = 3, ORD(дек) = 11.
```

Стандартный тип BOOLEAN – тоже перечислимый тип. Можно считать, что он имеет описание

```
BOOLEAN = (FALSE,TRUE)
```

Стандартные процедуры INC(x) и DEC(x) служат для присваивания переменной x соответственно следующего и предыдущего значения по сравнению с ее текущим значением.

17. ТИП ДИАПАЗОН

Если известно (или предполагается), что переменная будет принимать значения только внутри некоторого определенного диапазона величин, то этот факт можно выразить, описав так называемый тип диапазон. Допустим, например, что переменная i принимает значения только из диапазона от 1 до N включительно. Запишем следующие описания:

```
TYPE S = [1..N]

VAR i: S
```

(которые можно сократить до VAR i: [1..N]).

Каждый диапазон имеет некоторый <u>базовый</u> тип – тип его

значений. Все операции, определенные для базового типа, применимы также и к его диапазону. Единственное ограничение касается величин, которые могут быть присвоены переменным типа диапазон.

Синтаксис описания диапазона:

```
$      ТипДиапазон = [КвалИдент]
$           "[" КонстВыражение ".." КонстВыражение "]".
```

где выражения обозначают границы диапазона и должны содержать только константы. Приведем примеры описаний диапазонов.

```
ЛатБуква = ["A".."Z"]
Цифра = ["0"..."9"]
Индекс = INTEGER [1..100]
РабочийДень = [пн..пт]
```

Идентификатор, который может стоять перед квадратными скобками, указывает базовый тип диапазона. Этот идентификатор опускается, если базовый тип очевиден по виду границ. Но для целых это не всегда возможно. Если же идентификатор опущен в случае целых констант, то соблюдается следующее правило: если нижняя граница диапазона отрицательна, то базовым типом считается **INTEGER**, иначе **CARDINAL**. Отметим еще, что нельзя определять диапазон для типа **REAL**.

Тип диапазон имеет то преимущество, что дает дополнительную гарантию против некорректных присваиваний и может, следовательно, помочь в обнаружении ошибок. Следует, однако, указать, что проверки принадлежности величины диапазону происходят во время выполнения программы, поскольку такие ошибки не могут быть обнаружены лишь проверкой текста программы.

18. ТИП МНОЖЕСТВО

Каждый тип данных определяет некоторое множество значений. В том случае, когда тип S является типом множество (множественный тип), то область его значений – набор всевозможных множеств, составленных из элементов некоторого определенного базового типа B. Например, если базовый тип B – диапазон

```
B = [0..1]
```

и тип S описан как

```
S = SET OF B
```

то величинами типа S будут множества {}, {0}, {1}, {0,1}. Если базовый тип принимает n различных значений, то тип множество для

него будет иметь 2^N различных значений. Обозначение {} соответствует пустому множеству. В предшествующем разделе нам уже встречался стандартный множественный тип **BITSET**. Он определяется как

```
BITSET = SET OF [0..W-1]
```

где W – длина слова используемой ЭВМ. Операции объединения, пересечения и вычитания множеств, а также операция **IN** (определения принадлежности множеству) применимы не только к типу **BITSET**, а ко всем множественным типам. Для указания типа константного множества перед списком элементов, заключенным в фигурные скобки, должен стоять соответствующий идентификатор типа. Он может быть опущен в случае стандартного типа **BITSET**.

Синтаксис описания множественного типа:

```
$      ТипМножество = "SET" "OF" ПростойТип.
```

Синтаксис представления множеств как операндов выражений был приведен в разделе, посвященном стандартному типу **BITSET**. Напомним, что операнды множественного типа образуются посредством заключения списка элементов в фигурные скобки, перед которыми стоит идентификатор, указывающий тип множества (в случае типа **BITSET** он может быть опущен). Базовый тип множества должен быть или перечислением, или диапазоном. Кроме того, реализации Модулы ограничивают число элементов базового типа для множества длиной слова или некоторым небольшим числом, кратным ей. Обычно это число равно 16 или 32.

Хотя эти правила и ограничивают общность понятия "множество", тем не менее тип множество – мощное средство, позволяющее выразить операции над отдельными битами операнда на высоком уровне абстракции, основанном на хорошо знакомом математическом понятии. Для работы с множествами предлагаются две стандартные процедуры (они разворачиваются компилятором непосредственно в последовательность команд, без использования вызова): здесь s должна быть переменной, а x – выражением базового типа для s.

```
INCL(s,x)    включить элемент x в множество s

EXCL(s,x)    исключить элемент x из множества s
```

В заключение этой главы опишем одно приложение типа **BITSET**, не отражающее непосредственно понятие множества, но ставшее тем не менее очень важным и полезным. Оно касается представления информации о растре сканирующего дисплея. Эта информация называется битовой картой экрана, поскольку каждая отдельная точка экрана представляется отдельным битом в памяти ЭВМ, причем 1 обозначает черный, 0 – белый цвет (или наоборот). Такое битовое представление удобно описывается как массив элементов

типа **BITSET**. Предположим теперь, что мы должны представить в машине с длиной слова W экран дисплея с M строками, каждая из которых содержит N *точек*. (*Мы также считаем, что* N *кратно* W). Тогда соответствующее описание будет выглядеть так:

```
VAR КартаБитов: ARRAY [0..M*(N DIV W)-1] OF BITSET
```

Закрашивание точки (элемента изображения) с координатами x, y можно теперь выразить следующей процедурой:

```
PROCEDURE ЗакраситьЧерным(x,y: CARDINAL);
BEGIN
 INCL(КартаБитов[(N*y + x) DIV W],x MOD W)
END ЗакраситьЧерным
```

Процедура ЗакраситьБелым просто использовала бы **EXCL** вместо **INCL**. Мы предполагаем, что число N кратно W и что 0 <= x < N и 0 <= y < M. Если этого нельзя гарантировать, то в процедуру нужно включить соответствующие проверки. Очистка экрана эффективно реализуется присваиванием всем элементам массива пустого множества, вместо того чтобы оперировать с отдельными битами.

```
FOR i := 0 TO M*(N DIV W)-1 DO КартаБитов[i] := {} END
```

19. ТИП ЗАПИСЬ

Все элементы массива имеют один и тот же тип. Запись же, в отличие от массива, предоставляет возможность описать совокупность элементов как один объект, даже в случае их различных типов. Следующие примеры – это типичные случаи, которым хорошо подходит структуризация данных с помощью записей. Дата состоит из трех элементов: день, месяц, год. Описание некоторого человека может состоять из его имени, пола, идентифицирующего номера и даты рождения. Это выражается следующими описаниями типа:

```
Дата =    RECORD день: [1..31];
            мес: месяц;
            год: CARDINAL
          END

Человек = RECORD
            Фамилия,Имя,Отчество:
             ARRAY [0..23] OF CHAR;
            Мужчина: BOOLEAN;
            ИдентНомер: CARDINAL;
            Родился: Дата END
```


Запись дает возможность обращаться к ней и как к целому, и к ее отдельным элементам. Элементы записи называются также ее <u>компонентами</u> (или <u>полями</u>), а их имена называются <u>идентификаторами компонент</u>. Подобно массиву, где мы обозначаем i-й элемент массива a через a[i] (т.е. идентификатором массива, за которым следует индекс), компоненту f записи r мы будем обозначать через r.f, (т.е. после идентификатора записи через точку следует имя компоненты). Пусть даны переменные

```
d1,d2: Дата;
p1,p2: Человек;
Студент: ARRAY [0..99] OF Человек
```

Для них можно построить следующие обозначения:

```
d1.день
d2.мес
p1.Фамилия
p1.Родился
```

Эти примеры показывают, что поля в свою очередь могут быть структурированы. Аналогичным образом, записи сами могут быть элементами массива или компонентами другой записи, т.е. существует возможность строить иерархию структур. Следовательно, селекторы элементов могут образовывать последовательность, как это видно из дальнейших примеров. Многомерный массив, обсуждавшийся в разделе, посвященном массивам, теперь оказывается просто частным случаем структурной иерархии.

```
p1.Отчество[7]
p2.Рождения.год
Студент[23].ИдентНомер
Студент[k].Фамилия[0]
```

На первый взгляд может показаться, что запись – обобщение массива, поскольку снято требование совпадения типов всех элементов. Однако, с другой стороны, запись и более ограниченна, чем массив, поскольку компонента записи выбирается в соответствии с фиксированным идентификатором компоненты, а индекс, выбирающий элемент массива, может быть выражением, т.е. результатом предшествующих вычислений.

Важно отметить, что значение записи может представлять собой произвольную комбинацию значений ее компонент. Поэтому для типа Дата значение "день = 31" может сосуществовать с "мес = февр", хотя это значение и не является настоящей датой.

Синтаксис описания записи

```
$   ТипЗапись = "RECORD" ПослСписковКомпонент "END".
$   ПослСписковКомпонент =
```

```
$          СписокКомпонент {";" СписокКомпонент}.
$      СписокКомпонент = [СписИдент ":" Тип |
$          ВариантныйСписокКомпонент].
```

Синтаксис обозначения:

```
$      Обозначение = КвалИдент {Селектор}.
$      Селектор = "." Идентификатор |
$          "[" СписВыражений "]" | "^".
```

Примечание: Вариантные списки компонент и селектор ^ будут обсуждаться далее.

Чтобы выразить в удобном виде обработку элементов массива, был введен оператор цикла с параметром. Теперь, по аналогии, введем оператор, который служил бы для работы с компонентами записи в удобном виде. Поскольку каждая компонента записи имеет свой тип, то все они, вообще говоря, требуют различных действий. Следовательно, обработка записи не может быть выражена как применение одного и того же действия к различным компонентам. Поэтому более подходящая форма — последовательность отдельных операторов, включающих соответствующие компоненты записи. Поскольку все они принадлежат одной и той же записи, последовательность операторов можно снабдить заголовком присоединения, образовав тем самым оператор присоединения. Оператор присоединения задает переменную-запись, и он позволяет использовать имена ее компонент внутри последовательности операторов без предшествующего обозначения самой переменной и точки. Например, оператор

```
WITH d1 DO
  день := 10; мес := сент; год := 1981
END
```

эквивалентен последовательности операторов

```
d1.день := 10; d1.мес := сент; d1.год := 1981
```

Синтаксис оператора присоединения:

```
$      ОператорПрисоединения = "WITH" Обозначение "DO"
$          ПослОператоров "END".
```

Кроме того что оператор присоединения обеспечивает, возможно, более короткую форму записи, он может также давать и повышение эффективности в случае, если обозначение в заголовке присоединения содержит индексы. Индексные выражения вычисляются только один раз, поэтому рекомендуется соблюдать такое правило:

В последовательности операторов внутри оператора присоединения не должно содержаться присваиваний переменным, находящимся в заголовке присоединения, кроме, конечно, компонент присоединяемой записи.

Это правило подобно ограничению, запрещающему присваивания объектам, записанным в заголовке цикла с параметром. Программист должен отдавать себе отчет в том, что это правило — лишь рекомендация хорошего стиля программирования. Если правило нарушается, то последствия этого труднопредсказуемы, если вообще определены (* вероятно, будут зависеть от реализации. — Прим. перев.*). Более того, программист не должен надеяться, что компилятор обнаружит такие нарушения.

20. ЗАПИСИ С ВАРИАНТНЫМИ ЧАСТЯМИ

Тип запись обеспечивает и другого рода гибкость. Запись определенного типа может принимать различные вариантные формы. Имеется в виду, что число и типы компонент записи у разных переменных могут различаться, хотя все эти переменные будут одного типа. Очевидно, что такая гибкость ведет к трудноообнаружимым ошибкам в программе. В частности, теперь возможен такой случай, когда в некотором фрагменте программы будет предполагаться, что в переменной представлен один определенный вариант, в то время как на самом деле представлен совсем другой вариант. Следовательно, такой возможностью нужно пользоваться очень аккуратно.

Возможность образования вариантной записи иллюстрируется следующим примером:

```
Человек = RECORD
              Фамилия,Имя,Отчество: Строка;
              CASE мужчина: BOOLEAN OF
                TRUE:  ВоенноеЗвание: CARDINAL |
                FALSE: ДевичьяФамилия: Строка
              END;
              ИдентНомер: CARDINAL;
              Родился: Дата;
              CASE Положение: СемейноеПоложение OF
                одинок: |
                вБраке: Супруг: CARDINAL;
                        СколькоДетей: CARDINAL;
                        Свадьба: Дата |
                вдов:   Смерть: Дата
            END END
```

В этом примере содержится пять списков компонент, два из которых — вариантные списки. Они образованы согласно синтаксису:

```
$      ВариантныйСписокКомпонент = "CASE"
$            [Идентификатор] ":" КвалИдент "OF"
$            Вариант {"|" Вариант}
$            ["ELSE" ПослСписковКомпонент] "END".
$      Вариант = [СписокМетокВарианта ":"
$            ПослСписковКомпонент].
$      СписокМетокВарианта = МеткиВарианта
$            {"," МеткиВарианта}.
$      МеткиВарианта = КонстВыражение
$            [".." КонстВыражение].
```

Вариантный список состоит из заголовка выбора, за которым следуют списки компонент, разделенные символом "|". Вариантный список компонент имеет следующий смысл: можно работать лишь с тем списком компонент, который помечен текущим значением компоненты (не вариантной), указанной в заголовке выбора. Эта компонента называется селектором вариантов или дискриминантом.

Если обратиться к приведенному выше примеру и описать переменные p1 и p2 этого типа, то обозначение p1.ДевичьяФамилия допустимо, только если p1.Мужчина = FALSE, т.е. когда p1 представляет женщину. Аналогично, p2.Свадьба допустимо, только если p2.Положение = вБраке, т.е. p2 представляет человека, состоящего в браке. Селектор вариантов служит разделителем между разными вариантами и играет важную роль в уменьшении опасности ошибки обращения к недопустимым компонентам, как, например, к p1.Супруг, если p1.Положение = одинок. Вероятность ошибки еще уменьшается, если правильность обращения к компонентам с очевидностью следует из надлежащей структуры программы. Для таких целей вводится так называемый оператор выбора, который используется для разделения и различной обработки разных вариантов. Его можно рассматривать как обобщение условного оператора, учитывающего больше чем два случая. Синтаксис оператора выбора определяется так:

```
$      ОператорВыбора = "CASE" Выражение "OF"
$            Альтернатива {"|" Альтернатива}
$            ["ELSE" ПослОператоров] "END".
$      Альтернатива = [СписокМетокВарианта
$            ":" ПослОператоров].
```

Сходство синтаксиса оператора выбора с вариантным списком компонент весьма примечательно и отражает их тесную взаимосвязь. Это иллюстрируется следующим примером, который генерирует в доступной для чтения форме распечатку данных, представленных переменной

```
Лица: ARRAY [1..N] OF Человек
```

Обратите внимание на сходство структуры описания типа со структурой программы. Массив состоит из записей, содержащих вариантные части; оператор цикла с параметром содержит оператор присоединения, который в свою очередь содержит операторы выбора.


```
FOR i := 1 TO N DO
 WITH Лица[i] DO
  WriteString(Фамилия); Write(" ");
  WriteString(Имя); Write(" ");
  WriteString(Отчество);
  CASE мужчина OF
    TRUE:  WriteString(" мужчина, звание =");
      WriteCard(ВоенноеЗвание,4)|
    FALSE: WriteString(" женщина, девичья фамилия=");
        WriteString(ДевичьяФамилия)
  END;
  WriteCard(ИдентНомер,8);
  ПечатьДаты(Родился); WriteLn;
  CASE Положение OF
    одинок: WriteString(" одинок")|
    вБраке: WriteString(" в браке");
      WriteCard(Супруг,10);
      WriteCard(СколькоДетей,4);
      ПечатьДаты(Свадьба)|
    вдов:      WriteString(" вдов");
      ПечатьДаты(Смерть)
  END
 END
 WriteLn
END
```

Оператор выбора может, конечно, использоваться вне связи с вариантными записями. Но все же его следует применять только в таких ситуациях, когда величины, появляющиеся в качестве меток вариантов, расположены достаточно компактно. Проиллюстрируем это правило негативным примером; он показывает, как не нужно использовать оператор выбора.

```
CASE i*j OF
   1:  S1 |
  11:  S2 |
 121:  S3
ELSE   S4
END
```

Здесь предпочтение отдается записи с помощью условных операторов. В частности, ELSE следует резервировать для исключительных ситуаций, т.е. таких вариантов, доля которых мала среди всех возможных вариантов и которые редко случаются во время выполнения.

21. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ И УКАЗАТЕЛИ

Массивы, записи и множества имеют одно общее свойство: они статичны. Это подразумевает, что переменные с такой структурой сохраняют ее неизменной в течение всего времени их существования. Во многих приложениях это ограничение чересчур обременительно: в них требуется, чтобы переменные меняли не только свое значение, но и размер, составные части и структуру. Типичными примерами являются списки и деревья, которые растут и сокращаются динамически. Вместо того чтобы вводить дополнительные структуры типа "список" и "дерево", которых опять же оказалось бы недостаточно для других приложений, Модула предлагает базовое средство для построения произвольных структур. Это средство – указательный тип или указатель.

Любая сложная структура данных в конечном счете состоит из элементов, имеющих статическую структуру. Сами указатели, т.е. значения типа указатель, не структурированы, они лишь используются для установления связей между структурированными элементами, обычно называемыми узлами. Мы говорим также, что указатели связывают элементы или указывают на элементы. Очевидно, что различные указатели могут указывать на один и тот же элемент, тем самым обеспечивая возможность создавать сколь угодно сложные структуры и в то же время открывая массу возможностей для совершения трудноуловимых ошибок. Работа с указателями требует чрезвычайной тщательности и аккуратности.

Указатели в Модуле не могут указывать на произвольные переменные. Тип переменной, на которую ссылается указатель, должен быть задан в описании указателя, и говорят, что тип указателя подчинен типу объекта, на который он ссылается. Пример:

```
TYPE УкУзел = POINTER TO Узел:
VAR p0,p1: УкУзел
```

Здесь тип УкУзел (и, следовательно, переменные p0 и p1) подчинен типу Узел, т.е. эти переменные могут указывать на переменные только типа Узел. Однако сами такие переменные при описании p0 и p1 не возникают. Они создаются вызовом процедуры выделения памяти, имеющейся обычно в стандартном библиотечном модуле. Обычно модуль Storage (память) экспортирует процедуру Allocate (выделить память). Оператор Allocate(p0,SIZE(Узел)) создает переменную типа Узел и присваивает указатель на нее (указатель имеет тип УкУзел) переменной p0. О такой появившейся переменной говорят, что она создана (выделена) динамически. Она не имеет имени, и доступ к ней возможен только через указатель с использованием операции разыменования "^". В некоторых реализациях оператор Allocate(p0,SIZE(Узел)) может сокращаться как NEW(p0).

$ ТипУказатель = "POINTER" "TO" Тип.

Чрезвычайно мощным средством делает указатели то обстоятельство, что они могут указывать на переменные, которые сами содержат указатели. Это напоминает процедуры, вызывающие процедуры и тем самым вводящие рекурсию. Фактически указатели – средство реализации рекурсивно определенных структур данных (таких, как списки и деревья). Природа рекурсивной структуры данных очевидна из описания типа ее элементов.

Так же как любая рекурсия при активации процедуры должна когда-нибудь закончиться, любая рекурсия ссылок тоже где-нибудь заканчивается. Роль условного оператора в прекращении процедурной рекурсии берет на себя в рекурсивных структурах данных специальное значение указателя NIL, завершающее рекурсию ссылок. NIL не указывает ни на какой объект. Можно представить себе, что каждый указательный тип описан как запись с двумя вариантами: один вариант указывает на объект заданного типа, а другой не указывает на объекты, т.е. имеет NIL в качестве единственного значения. Из сказанного очевидно, что обозначение p^ не должно вычисляться, если p = NIL.

Выделим следующие существенные моменты.

1. Каждый указательный тип подчинен некоторому другому типу; значения указательного типа – ссылки на объекты того типа, которому он подчинен.
2. Переменная, доступная по ссылке, не имеет имени, и доступ к ней возможен только через посредство указателей.
3. Переменные, доступные по ссылке, динамически создаются процедурой выделения памяти, которая заносит указатель на эту переменную в p.
4. Указательная константа NIL принадлежит всем указательным типам и не указывает ни на какой объект.
5. Переменная, адресуемая указателем p, имеет обозначение p^. Чтобы обозначение p^ имело смысл, p не должна иметь значение NIL.

Списки, называемые также линейными списками, характеризуются тем, что они состоят из узлов, каждый из которых имеет (в точности одну) компоненту – указатель на узел того же типа, что подразумевает рекурсию. Типы узлов – это, как правило, записи; для списков описание типа принимает следующий характерный вид:

```
УкСписка = POINTER TO УзелСписка
УзелСписка =
 RECORD
  ключ: CARDINAL;
  Данные:...
  следующий: УкСписка
 END
```

"Данные" в действительности могут быть представлены любым числом компонент, представляющих данные, принадлежащие данному узлу. Ключ – часть этих данных: он упоминается здесь отдельно, поскольку принято связывать уникальный идентифицирующий ключ с каждым элементом, а также потому, что ключ будет использоваться в последующих примерах действий над списками. Существенной частью записи является, однако, компонента "следующий", которая, очевидно, помечена так потому, что это указатель на следующий элемент списка. Прямая рекурсия в описании типа (минуя POINTER TO) запрещена по той простой причине, что она не могла бы быть завершена. Приведенное выше описание нельзя сократить до

```
Список =
 RECORD
  ключ: CARDINAL:...
  следующий: Список
 END
```

Предположим теперь, что список доступен в программе через его первый элемент, на который ссылается указательная переменная.

```
первый: УкСписка
```

Пустой список представляется пустой ссылкой: первый = NIL. Удлинение списка проще всего осуществляется вставками новых элементов в его начало. Приведенная ниже последовательность операторов вставляет новый элемент в список (p – произвольная переменная типа УкСписка).

```
Allocate(p,SIZE(УзелСписка));
WITH p^ DO
 (* присвоить значения ключу и данным *)
 следующий := первый
END;
первый := p
```

Построив список последовательной вставкой узлов, мы можем пожелать <u>найти в списке узел</u>, ключ которого равен заданной величине x. Очевидно, нужно использовать повторение: для этого подходит цикл с условием продолжения, поскольку мы не знаем заранее количества узлов, а следовательно, повторений. Мудрая предосторожность – предусмотреть в алгоритме возможность пустого списка.

```
p := первый;
WHILE (p # NIL) & (p^.ключ # x) DO
 p := p^.следующий
END;
IF p # NIL THEN найден END
```


Обратите внимание на использование в алгоритме правила вычисления логического произведения a&b. Если оказывается, что a ложно, то b не вычисляется. Если бы это было не так, то могло бы случиться, что логический сомножитель (p^.key # x) вычислялся при p = NIL, а это запрещено.

<u>Удаление</u> элемента особенно просто для первого узла:

```
p := первый;
первый := p^.следующий;
Deallocate(p,SIZE(УзелСписка))
```

Мы здесь считаем, что процедура **Deallocate** импортируется из того же модуля, что и **Allocate**. **Deallocate** вновь освобождает память, выделенную ранее под переменную p. При пользовании этой процедурой важна крайняя осторожность: если указатель на уничтожаемую переменную был присвоен другим переменным, то теперь они будут указывать на несуществующий объект. Программисту не следует надеяться, что эта ошибка будет автоматически обнаружена системой.

Другая часто встречающаяся динамическая структура – это <u>дерево</u>. Ее особенность заключается в том, что каждый узел имеет n компонент-указателей. Число n называют степенью дерева. Наиболее частым и в некотором смысле оптимальным вариантом является двоичное дерево с n = 2. Приведем соответствующие описания.

```
УкДерева =
 POINTER TO УзелДерева;

УзелДерева =
 RECORD
  ключ: CARDINAL;
  Данные:...
  левый,правый: УкДерева
 END
```

Роль переменной "первый", как было в случае списков, будет выполнять переменная, называемая

```
корень: УкДерева
```

При корень = NIL считается, что дерево пустое. Деревья часто используются для представления совокупностей данных с возрастающим порядком ключей, причем таким образом, чтобы поиск элементов по ключу был очень эффективным. Приведем последовательность операторов, осуществляющих <u>поиск в упорядоченном двоичном дереве</u>. Заслуживает внимания его сходство с двоичным поиском в упорядоченном массиве. Как и раньше, p – переменная типа УкДерево.

```
p := корень;
WHILE (p # NIL) & (p^.ключ = x) DO
 IF p^.ключ < x THEN p := p^.правый
 ELSE p^.левый
END END
IF p # NIL THEN найден END
```

Этот пример является циклической версией поиска по дереву. Теперь приведем рекурсивную версию. Она, кроме того, дополнена так, что когда узла с данным значением ключа не существует в дереве, то он создается и вставляется на соответствующее место.

```
PROCEDURE Поиск(VAR p: УкДерева; x: CARDINAL): УкДерева;
BEGIN
 IF p # NIL THEN
  IF p^.ключ < x THEN
   RETURN Поиск(p^.правый,x)
  ELSIF p^.ключ > x THEN
   RETURN Поиск(p^.левый,x)
  ELSE
   RETURN p
  END
 ELSE (* узел не найден, вставка узла *)
  Allocate(p,SIZE(УзелДерева));
  WITH p^ DO
   ключ := x; левый := NIL; правый := NIL
  END;
  RETURN p
 END
END Поиск
```

Теперь вызов процедуры-функции поиск(корень,x) осуществляет поиск по дереву, представленному переменной "корень".

На этом закончим примеры действий над списками и деревьями, иллюстрирующие работу с указателями. У списков и деревьев все узлы имеют один и тот же тип. Однако указатели позволяют создавать структуры и более общего вида, состоящие из узлов различного типа. Типично для всех этих структур то, что все их узлы описываются как записи, так что запись становится особенно полезной структурой при использовании ее совместно с указателями.

Создание и уничтожение узлов осуществляются стандартными процедурами выделения и освобождения памяти, которые являются частью системного механизма управления памятью. Иногда может оказаться более эффективной самостоятельная работа программы с памятью без обращения к системным средствам. Это легко реализовать, заведя в программе для каждого типа узла список и занося освобождаемые узлы в соответствующие списки. Этот список просматривается всякий раз, когда требуется новый узел.


```
PROCEDURE НовУзел(VAR p: УкУзел);
BEGIN
 IF свободные = NIL THEN Allocate(p,SIZE(Узел))
 ELSE p := свободные; свободные := p^.следующий
 END
END НовУзел
```

Недостаток такого "персонального управления памятью" в том, что при наличии нескольких списков свободных узлов не происходит перераспределения памяти между ними.

22. ПРОЦЕДУРНЫЕ ТИПЫ

До сих пор мы рассматривали процедуры исключительно как фрагменты программ, т.е. тексты, определяющие действия, которые должны быть выполнены над переменными с числовыми, логическими, символьными и другими значениями. Однако процедуры можно рассматривать и как объекты, которые можно присваивать переменным. При таком подходе описание процедуры представляет собой особую разновидность описания константы, значением которой и является процедура. Если мы в дополнение к константам введем переменные, то станет возможным описывать типы, значениями которых будут процедуры. Такие типы называются процедурными типами.

Описание процедурного типа задает число и типы параметров, а также, если это функция, тип результата. Например, процедурный тип с одним аргументом типа REAL и с таким же результатом описывается как

```
Func = PROCEDURE(REAL): REAL
```

а тип с двумя аргументами типа CARDINAL –

```
Proc2 = PROCEDURE(CARDINAL,CARDINAL)
```

Общий синтаксис описания процедурного типа:

```
$   ТипПроцедура = "PROCEDURE" [СписокФормТипов].
$   СписокФормТипов = "(" [["VAR"] ФормТип
$        {"," ["VAR"] ФормТип}] ")"
$        [":" КвалИдент].
```

Если мы теперь опишем, например, переменные

```
f: Func
p: Proc2
```

то возможны такие присваивания:

```
f := sin; p := WriteCard
```

После этого вызов p(x,6) станет эквивалентен WriteCard(x,6), а f(x) станет эквивалентен sin(x).

Теперь можно описать процедуры, параметрами которых в свою очередь являются процедуры. Рассмотрим, например, следующую задачу: нужно над каждым элементом двоичного дерева проделать некоторое действие, т.е. выполнить процедуру. Удобно записать решение в виде рекурсивной процедуры, описывающей обход дерева и вызывающей для каждого обходимого узла требуемую процедуру. Последняя получается как параметр и называется поэтому формальной процедурой.

```
PROCEDURE ОбходДерева(p: УкДерево; Q: Proc2);
BEGIN
  IF p # NIL THEN
   ОбходДерева(p^.левый,Q);
   Q(p^.ключ,6);
   ОбходДерева(p^.правый,Q)
  END
END ОбходДерева
```

Если мы теперь запишем

```
ОбходДерева(корень,WriteCard)
```

то значения ключей всех узлов будут выведены в порядке, задаваемом деревом. Эта же процедура может использоваться для вывода ключей в восьмеричном виде, например вызовом

```
ОбходДерева(корень,WriteOct)
```

Из этих примеров должно быть ясно, что, хотя процедурные типы редко используются, они представляют собой мощное средство программирования.

В заключение укажем, что процедуры, присваиваемые переменным или передаваемые как параметры, не могут быть описаны как локальные в некоторой процедуре. Они также не могут быть стандартными процедурами (такими, как ODD, INCL и т.д.).

Часть 4

23. МОДУЛИ

Модули – самая важная черта, отличающая язык Модула-2 от ее предшественника Паскаля. Мы уже встречались с модулями, поскольку каждая программа – это модуль. Однако большинство программ разбивается на несколько модулей, причем каждый модуль содержит константы, переменные, процедуры и, возможно, типы. Из модуля M можно обращаться к объектам, описанным в других модулях, если эти объекты сделаны доступными, т.е. импортированы в модуль M. В примерах предыдущих разделов мы обычно импортировали процедуры ввода и вывода из модулей, содержащих наборы часто используемых процедур. Эти процедуры – в действительности часть нашей программы, даже если мы их и не программировали, и они текстуально отделены.

Ключевая идея состоит в том, что модули могут храниться в "библиотеке" программ и к ним должно производиться автоматическое обращение при загрузке и выполнении программы, непосредственно написанной программистом. Таким образом, оказывается возможным заранее заготовить набор часто используемых операций (таких, как ввод и вывод) и избежать необходимости повторного программирования этих функций всякий раз, когда они потребуются. Более сложные реализации языка идут в этом отношении дальше и предлагают средство, называемое раздельной компиляцией. Это означает, что модули запоминаются в библиотеке программ не в виде исходных текстов, а в откомпилированной форме. При загрузке программы, компилируемая (главная) программа объединяется (связывается) с теми ранее откомпилированными модулями, из которых она импортирует объекты. В этом случае компилятор во время трансляции импортирующей программы должен также иметь доступ к описаниям объектов тех ранее откомпилированных модулей, из которых происходит импорт. Эта черта отличает раздельную компиляцию от независимой компиляции, присутствующей в типичных реализациях Фортрана, Паскаля и ассемблера.

Любой вспомогательный модуль может в свою очередь импортировать объекты из других модулей. Следовательно, программа может представлять собой целую иерархию модулей. Говорят, что главная программа имеет наивысший уровень, а те

модули, которые совсем не импортируют объектов – наинизший. Обычно программист даже не имеет представления об этой иерархии, поскольку его программа импортирует объекты из модулей, которые он сам не писал и, следовательно, не знает их импорта, а значит, и иерархии модулей, лежащих ниже. Но тем не менее в принципе программа не что иное, как текст, написанный самим программистом и расширенный текстами импортируемых модулей. Эти расширения обычно очень велики (даже если прямой импорт состоит лишь из нескольких процедур вывода). В принципе непрямой импорт включает в себя полностью все окружение, или операционную систему. В вычислительных системах, ориентированных на одного пользователя, фактически не требуется ничего, что прямо или косвенно не импортировалось бы главной программой. Однако некоторые модули, такие, как базовая система ввода-вывода и файловая система, могут запрашиваться всеми программами и фактически становятся резидентными, а следовательно, их можно считать операционной системой.

Принципиальное соображение, лежащее в основе разбиения программы на модули (не считая преимуществ использования модулей, написанных другими программистами), – необходимость установления иерархии абстракций. Например, во встречавшихся ранее примерах импорта процедур ввода-вывода мы просто хотели их иметь и нам не нужно было знать (мы даже не хотели узнавать), как конкретно работают эти процедуры. Абстрагирование означает "пренебрежение" содержимым и тем самым игнорирование конкретных деталей. Каждый модуль представляет собой абстракцию, если рассматривать его "снаружи". Мы хотим даже большего: не просто игнорировать детали внутренности модуля, а скрыть их. Первая причина такого желания заключается в том, что если внутренность модуля защищена от внешнего доступа, то можно гарантировать его правильное функционирование, ограничивая тем самым область поиска ошибки в случае неверного функционирования программы. Вторая, но не менее важная причина – сделать возможным изменение (улучшение) внутренней структуры импортируемого модуля без изменения (и/или перекомпиляции) импортирующего модуля. Такая эффективная декомпозиция модулей просто необходима при разработке больших программ, в особенности если модули разрабатываются разными людьми и если мы рассматриваем операционную систему как секцию низкого уровня в иерархии программных модулей. Без декомпозиции любое изменение или исправление в операционной системе или библиотеке модулей было бы фактически невозможным.

Прямым следствием требования декомпозиции является необходимость текстуального разделения внешней спецификации модуля и его внутренних деталей. Внешняя спецификация модуля – это информация об объектах, которые могут импортироваться другими модулями. Внутренние детали – те части модуля, которые должны быть скрыты и защищены. В Модуле-2 декомпозиция достигается разбиением модуля на раздел определений и раздел реализации. Раздел определений содержит описания экспортируемых объектов, и он должен рассматриваться как приставка к разделу реализации. При импорте модуля достаточно иметь доступ только к его разделу определений; раздел реализации остается в полном распоряжении разработчика модуля. До тех пор пока он меняет лишь раздел реализации (разумным образом), ему не нужно ничего сообщать о своих действиях тем, кто использует модуль. Оба раздела компилируются отдельно и поэтому называются единицами компиляции.

Завершая это введение в понятие модуля, потребуем, чтобы всякая разумная реализация обеспечивала полный контроль совместимости типов между объектами независимо от того, описаны они в одном и том же или в разных модулях, т.е. границы модуля не должны ограничивать механизм контроля типов в компиляторе. Однако программисту следует понимать, что и такая проверка не дает абсолютной защиты от ошибок. В конце концов она касается только формальных, синтаксических аспектов и не затрагивает семантическую правильность. Она не обнаружила бы, например, ошибочную замену алгоритма вычисления синуса алгоритмом для косинуса. Но ведь нельзя считать обязанностью компилятора защиту программистов от козней их коллег.

24. РАЗДЕЛ ОПРЕДЕЛЕНИЙ И РАЗДЕЛ РЕАЛИЗАЦИИ

Раздел определений модуля называется модулем определений. Он содержит описания экспортируемых идентификаторов. Они могут обозначать объекты любого вида, но следует упомянуть несколько дополнительных правил.

Модуль – часть полного текста программы. Следовательно, переменные, описанные в модуле определений, являются глобальными в том смысле, что они существуют в течение всего периода выполнения программы, хотя видимы и доступны только в тех модулях, которые их импортируют. В остальных модулях они невидимы.

Описания процедур в модуле определений состоят из одних заголовков. Тела процедур находятся в соответствующем модуле реализации.

Если тип описан в модуле определений, то все детали описания видимы в импортирующих модулях. Если описан тип перечисление или тип запись, то экспорт его имени автоматически ведет соответственно к экспорту имен констант перечисления или имен компонент. В противовес такому прозрачному экспорту существует еще скрытый экспорт типов. Экспорт получается скрытым, если в модуле определений описано просто имя типа; в этом случае детали описания скрыты в модуле реализации. Скрытый экспорт возможен только для указательного типа. Этот тип, однако, весьма важен потому, что указательный тип подчинен другому типу (обычно записи), который может быть скрыт. Далее следуют примеры.

иллюстрирующие упрятывание деталей описания типов, известное также под названием "абстракции данных".

Следующий простой пример проясняет существенные свойства модулей, хотя обычно модули – значительно большие фрагменты программы, и они содержат более длинный список описаний. В этом примере экспортируются две процедуры занести и взять, соответственно добавляющие в буфер и извлекающие из него данные, причем сам буфер скрыт. Фактически доступ к буферу может осуществляться только через эти две процедуры, и, следовательно, может быть гарантировано его надлежащее функционирование.

```
DEFINITION MODULE Буфер;
 VAR непуст,неполон: BOOLEAN;
 PROCEDURE занести(x: CARDINAL);
 PROCEDURE взять(VAR x: CARDINAL);
END Буфер.
```

Этот раздел определений содержит всю информацию о буфере, которую, как предполагается, должен знать пользователь модуля. Детали его функционирования, его реализации содержатся в соответствующем модуле реализации.

```
IMPLEMENTATION MODULE Буфер;
 CONST N = 100;
 VAR in,out: [0..N-1];
  n: [0..N];
  буф: ARRAY [0..N-1] OF CARDINAL;

 PROCEDURE занести(x: CARDINAL);
 BEGIN
  IF n < N THEN
   буф[in] := x; in := (in+1) MOD N;
   n := n + 1; неполон := n < N; непуст := TRUE
  END
 END занести;

 PROCEDURE взять(x: CARDINAL);
 BEGIN
  IF n > 0 THEN
   x := буф[out]; out := (out+1) MOD N;
   n := n - 1; непуст := n > 0; неполон := TRUE
  END
 END взять;

BEGIN (*инициализация*)
 n := 0; in := 0; out := 0;
 непуст := FALSE; неполон := TRUE
END Буфер.
```


Этот пример реализует дисциплину очереди (первым пришел – первым обслужен). Этот факт совсем не следует из содержания модуля определений: обычно семантика упоминается в виде комментария или при помощи какой-либо другой формы документирования. Такие комментарии объясняют то, что делает модуль, но не то, как это делается. Следовательно, различные модули реализации могут иметь один и тот же модуль определений. Различия могут заключаться в конкретном механизме реализации; например, буфер представлен как связанный список элементов, а не массив (причем элементы под буфер выделяются по мере необходимости, а значит, размер буфера не ограничен). Но различия могут быть и в семантике. Следующая программа реализует не очередь, а стек (последним пришел – первым обслужен) и тем не менее к ней подходит тот же модуль определений. Любые изменения в семантике требуют соответствующей адаптации пользовательских модулей, следовательно, такие изменения должны осуществляться с предельной осторожностью.

```
IMPLEMENTATION MODULE Буфер;
  ..............

 PROCEDURE занести(x: CARDINAL);
 BEGIN
  IF n < N THEN
   буф[n] := x; n := n + 1;
   неполон := n < N; непуст := TRUE
  END
 END занести;

 PROCEDURE взять(x: CARDINAL);
 BEGIN
  IF n > 0 THEN
   n := n - 1; x := буф[n];
   непуст := n > 0; неполон := TRUE
  END
 END взять;

BEGIN
 n := 0; непуст := FALSE; неполон := TRUE
END Буфер.
```

Очевидно, что условие "непуст" является предусловием для процедуры взять, а "неполон" – для занести. На этом завершим вводный пример.

Синтаксис модуля определений:

```
$   МодульОпределений =
$       "DEFINITION" "MODULE" Идентификатор ";"
$       {Импорт} {Определение}
$       "END" Идентификатор ".".
```

```
$      Определение = "CONST" {ОписаниеКонстанты ";"} |
$            "TYPE" {Идентификатор ["=" Тип] ";"} |
$            "VAR" {ОписаниеПеременной ";"} |
$            ЗаголовокПроцедуры ";".
```

Синтаксис разделов реализации совпадает с синтаксисом главной программы, за исключением ключевого слова **IMPLEMENTATION**, добавляемого в начале для указания того, что для этого модуля существует модуль определений, описания которого автоматически считаются принадлежащими и данному модулю.

```
$      ПрограммныйМодуль =
$            "MODULE" Идентификатор [Приоритет] ";"
$            {Импорт} Блок Идентификатор.
$      ЕдиницаКомпиляции = МодульОпределений |
$            ["IMPLEMENTATION"] ПрограммныйМодуль.
```

Как раздел определений, так и раздел реализации могут содержать списки импорта (один или несколько). В модуль определений следует импортировать лишь те объекты, которые действительно требуются в нем самом. Это уменьшает его зависимость от других модулей.

```
$      Импорт = ["FROM" Идентификатор]
$            "IMPORT" СписИдент ";".
```

Идентификатор, следующий за ключевым словом **FROM**, – имя модуля, из которого импортируются объекты. Без использования такого квалифицирования мы можем импортировать только имена модулей (о том, как ослабить это правило, будет сказано в разделе, посвященном локальным модулям). Если импортируется имя модуля, то автоматически импортируются все имена, находящиеся в его разделе определений. Однако в этом случае они должны квалифицироваться именем модуля подобно компоненте записи. Например, если модуль M экспортирует объекты a, b, c, то запись **IMPORT M**, встретившаяся в модуле N, означает, что к объектам можно обращаться с помощью обозначений M.a, M.b, M.c. Это средство позволяет импортировать из различных модулей объекты с совпадающими именами и при этом избежать конфликта имен. В данном случае M действует как <u>квалифицирующий идентификатор</u>.

Возможность оформить внешние связи модуля в виде раздела определений и скрыть подробности функционирования в разделе реализации особенно удобна при организации библиотек подпрограмм. Такие наборы стандартных подпрограмм имеются в любой программной среде. Они, как правило, включают программы ввода-вывода, операции работы с файлами и процедуры вычисления математических функций. Хотя в Модуле и не существует жестких стандартов, тем не менее модули **InOut**, **RealInOut**, **LineDrawing**, **MathLib0** и **Streams** (или их эквиваленты) можно считать стандартными, имеющимися во всех реализациях языка. Эти модули вводятся в книге далее. Здесь же в качестве первого примера приведем раздел определений модуля **MathLib0**.

```
DEFINITION MODULE MathLib0:
  PROCEDURE sqrt(x: REAL): REAL;
  PROCEDURE exp(x: REAL): REAL;
  PROCEDURE ln(x: REAL): REAL;
  PROCEDURE sin(x: REAL): REAL;
  PROCEDURE cos(x: REAL): REAL;
  PROCEDURE arctan(x: REAL): REAL;
  PROCEDURE real(x: INTEGER): REAL;
  PROCEDURE entier(x: REAL): INTEGER;
END MathLib0.
```


25. РАЗБИЕНИЕ ПРОГРАММЫ НА МОДУЛИ

Под качеством программы подразумеваются многие аспекты, и эта сторона дела очень расплывчата и неуловима. Пользователь может судить о качестве программы по ее эффективности, надежности или удобству диалога. Эффективность в принципе можно выразить на языке цифр, но, например, удобство использования – это скорее дело личного вкуса, и в преобладающем большинстве случаев способ взаимодействия с программой считается удобным, если он общепринят. Разработчик может судить о качестве по ясности и понятности программы, свойствам тоже весьма расплывчатым. Однако если некоторое свойство и не может быть выражено на языке точных цифр, то это еще не причина считать его несущественным. На самом деле ясность программы – чрезвычайно важная характеристика, ведь демонстрация (доказательство?) правильности в конечном счете сводится к убеждению человека в том, что программа надежна. Как нам достигнуть этой цели? Ведь, в конце концов, сложные задачи по самой своей природе требуют сложных алгоритмов, которые подразумевают гигантское множество деталей. И эти детали вырастают в целый лес, полный привидений.

Единственное спасение – организация соответствующей структуры программы. Программа должна быть разбита на части так, чтобы их можно было рассматривать по отдельности, почти полностью независимо друг от друга. На самом нижнем уровне элементами структуры являются операторы, на следующем – процедуры, а на верхнем уровне – модули. Параллельно структуризации программ идет структуризация данных. На нижнем уровне она осуществляется при помощи массивов, записей и т.д., а на последующих уровнях это происходит за счет объединения переменных с процедурами и модулями. Сущность программирования заключается в поиске правильной (или, по крайней мере, подходящей) структуры, и опытный программист – как раз тот человек, который обладает интуитивной способностью найти такую структуру именно на этапе

первоначальной разработки программы, а не в процессе ее постепенных улучшений и модификаций. Однако программист, имеющий смелость поменять структуру своей программы в случае обнаружения более удачного решения, гораздо лучше того, кто занимается подчисткой и вылизыванием программы, в основе которой лежит явно неадекватная структура, поскольку это ведет к программному продукту, "непонятному" ни для кого, кроме его создателя (а в конечном счете и для него самого).

Хотя и не существует рецепта определения оптимальной структуры программы, тем не менее выработались некоторые критерии для руководства процессом поиска хорошей структуры и для предотвращения получения плохой. Основное правило требует производить разбиение так, чтобы связи между частями были простыми и "слабыми". Возможно, наипростейшим критерием слабости связи (называемой также интерфейсом) между двумя частями является число объектов, в ней участвующих. В частности, интерфейс двух модулей описывается с помощью списков импорта модуля, и мерой слабости интерфейса можно считать число импортируемых объектов. Следовательно, мы должны выбрать такое разбиение на модули, которое делает списки импорта короткими. Естественно, оптимум найти трудно, поскольку списки импорта были бы самыми короткими, т.е. совсем исчезли, если бы мы всю программу объединили в один модуль, что, очевидно, нежелательно.

Способность модулей как наибольших структурных единиц разделять программу на относительно независимые части определяется их свойством скрывать детали и тем самым образовывать новый уровень абстракции. Это свойство используется по-разному. Можно выделить следующие типичные случаи:

1. Модуль связывает две формы представления данных и содержит набор процедур, которые осуществляют преобразование между этими формами. Типичный пример — перевод чисел из атомарной, неделимой формы в последовательность десятичных цифр и наоборот. Такие модули не содержат собственных данных, обычно это просто наборы процедур.

2. Основное содержание модуля — некоторая совокупность данных. Подробности представления данных скрыты, и доступ к ним осуществляется лишь с помощью экспортируемых модулем процедур. Примером может служить модуль, содержащий множество отдельных элементов, организованных так, что доступ к ним по ключу происходит очень быстро. Другой пример — модуль, скрытой совокупностью данных которого является дисковая память: он скрывает специфические детали, необходимые для работы котроллера диска.

3. Модуль экспортирует тип данных и набор связанных с этим типом операций. В Модуле-2 обычно такой модуль экспортирует один или несколько типов в скрытом режиме (иногда эти типы называются приватными). Тем самым скрывается структура типа и детали выполнения операций. Такое упрятывание информации позволяет дать гарантию истинности постулированных инвариантных свойств каждой переменной приватного типа. Отличие от случая 2 состоит в том, что здесь переменные приватного типа описаны в модуле пользователя, а в случае 2 сама переменная скрыта. Типичные примеры — очередь и стек. Возможно, наиболее удачный пример такой абстракции данных — это последовательный файл, называемый также потоком.

Такая классификация не абсолютна. Да абсолютной и не может быть, так как все указанные случаи могут совмещаться. Модуль InOut, уже использовавшийся в примерах программ, совмещает черты модулей первого и второго видов: скрывает детали представления чисел и их преобразования, а также скрывает две потоковые переменные In и Out. Тем не менее мы можем сформулировать несколько правил, служащих руководством при разработке модулей.

1. Число импортируемых идентификаторов должно быть небольшим.
2. Правило 1 особенно важно для модулей определений.
3. Экспорт переменных следует считать исключением, все импортируемые переменные следует использовать лишь для чтения.

Завершим эту главу примером, относящимся к третьему случаю. Пусть нашей целью является разработка генератора перекрестных ссылок для программ, написанных на Модуле. Более точно, задачей программы является чтение текста и получение его распечатки (1), дополненной номерами строк, а также таблицы всех встретившихся слов (идентификаторов) в алфавитном порядке (2), причем для каждого слова должен печататься список номеров строк, в которых оно встречается. Кроме того, комментарии и строки должны пропускаться (т.е. не печататься содержащиеся в них слова), не должны также печататься ключевые слова и стандартные идентификаторы.

В этой задаче можно выделить такие функции: просмотр исходного текста (с пропуском ненужных участков), запоминание встретившихся идентификаторов и их последующая печать в виде таблицы. Первая функция выполняется модулем главной программы, а вторая — вспомогательным модулем, который скрывает множество данных и обеспечивает доступ к ним через две процедуры: Записать (т.е. занести слово) и Распечатать (т.е. печатать требуемую таблицу). Третий модуль предназначен для перевода чисел из внутреннего представления в последовательности десятичных цифр. Эти три основных модуля называются соответственно XREF, РаботаСТаблицей и InOut.

Опишем сначала главную программу XREF, которая просматривает исходный текст. Для распознавания ключевых слов используется двоичный поиск. Множество данных имеет тип Таблица. Он импортируется из модуля РаботаСТаблицей в скрытом режиме.

```
DEFINITION MODULE РаботаСТаблицей;
 CONST ШиринаСтроки = 80; ДлинаСлова = 24;
 TYPE Таблица;
 VAR переполнение: CARDINAL;
      (* >0 означает, что таблица полна *)

 PROCEDURE Инициализировать(VAR t: Таблица);
 PROCEDURE Записать(t: Таблица;
              VAR x: ARRAY OF CHAR; n: INTEGER);
  (* занести пару x,n в таблицу
     строка x должна заканчиваться пробелом *)
 PROCEDURE Распечатать(t: Таблица)
END РаботаСТаблицей.


MODULE XREF;
 FROM InOut IMPORT
  Done,EOL,OpenInput,OpenOutput,Read,Write,WriteCard,
  WriteString,CloseInput,CloseOutput;
 FROM РаботаСТаблицей IMPORT
  ДлинаСлова,Таблица,переполнение,
  Инициализировать,Записать,Распечатать;

 TYPE Alfa = ARRAY [0..9] OF CHAR;
 CONST N = 45; (* число ключевых слов *)
 VAR ch: CHAR;
  i,k,l,m,r,номстр: CARDINAL;
  T: Таблица;
  идент: ARRAY [0..ДлинаСлова-1] OF CHAR;
  ключ: ARRAY [1..N] OF Alfa;

 PROCEDURE Копировать;
 BEGIN Write(ch); Read(ch);
 END Копировать;

 PROCEDURE Заголовок;
 BEGIN номстр := номстр + 1;
  WriteCard(номстр,5); Write(" ")
 END Заголовок;
BEGIN Инициализировать(T);
 ключ[1]:= "AND";          ключ[2]:= "ARRAY";
 ключ[3]:="BEGIN";         ключ[4]:= "BITSET";
 ключ[5]:= "BOOLEAN";      ключ[6]:="BY";
 ключ[7]:= "CASE";         ключ[8]:= "CARDINAL";
 ключ[9]:="CHAR";          ключ[10]:="CONST";
 ключ[11]:="DIV";          ключ[12]:="DO";
 ключ[13]:="ELSE";         ключ[14]:="ELSIF";
 ключ[15]:="END";          ключ[16]:="EXIT";
 ключ[17]:="EXPORT";       ключ[18]:="FALSE";
```



```
 ключ[19]:="FOR";          ключ[20]:="FROM";
 ключ[21]:="IF";           ключ[22]:="IMPORT";
 ключ[23]:="IN";           ключ[24]:="INTEGER";
 ключ[25]:="LOOP";         ключ[26]:="MOD";
 ключ[27]:="MODULE";       ключ[28]:="NOT";
 ключ[29]:="OF"            ключ[30]:="OR";
 ключ[31]:="POINTER";      ключ[32]:="PROCEDURE";
 ключ[33]:="QUALIFIED";    ключ[34]:="RECORD";
 ключ[35]:="REPEAT";       ключ[36]:="RETURN";
 ключ[37]:="SET";          ключ[38]:="THEN";
 ключ[39]:="TO";           ключ[40]:="TRUE";
 ключ[41]:="TYPE";         ключ[42]:="UNTIL";
 ключ[43]:="VAR";          ключ[44]:="WHILE";
 ключ[45]:="WITH";

 OpenInput("MOD");
 IF NOT Done THEN HALT END;
 OpenOutput("XREF");
 номстр := 0; Read(ch);
 IF Done THEN Заголовок;
  REPEAT
   IF (CAP(ch) >= "A") & (CAP(ch) <= "Z") THEN
    k := 0;
    REPEAT идент[k] := ch; k := k + 1; Копировать
    UNTIL (ch < "0") OR
      (ch > "9") & (CAP(ch) < "A") OR
      (CAP(ch) > "Z");
    l := 1; r := N; идент[k] := " ";
    REPEAT m := (l+r) DIV 2; i := 0;(*двоичный поиск*)
     WHILE (идент[i]=ключ[m,i])&(идент[i]>" ") DO
      i := i + 1
     END;
     IF идент[i] <= ключ[m,i] THEN r := m - 1 END;
     IF идент[i] >= ключ[m,i] THEN l := m + 1 END;
    UNTIL l > r;
    IF l = r + 1 THEN Записать(t,идент,номстр) END
   ELSIF (ch >= "0") & (ch <= "9") THEN
    REPEAT Копировать
    UNTIL ((ch<"0")OR(ch>"9")) & ((ch<"A")OR(ch>"z"))
   ELSIF ch = "(" THEN
    Копировать;
    IF ch = "*" THEN (*комментарий*)
     REPEAT
      REPEAT
       IF ch = EOL THEN
        Копировать; Заголовок
       ELSE Копировать
       END
      UNTIL ch = "*";
```

```
      Копировать
     UNTIL ch = ")";
     Копировать
    END
   ELSIF ch = "'" THEN
    REPEAT Копировать UNTIL ch = "'";
    Копировать
   ELSIF ch = '"' THEN
    REPEAT Копировать UNTIL ch = '"';
    Копировать
   ELSIF ch = EOL THEN
    Копировать;
     IF Done THEN Заголовок END
   ELSE Копировать
   END
  UNTIL NOT Done OR (переполнение # 0)
 END;
 IF переполнение > 0 THEN
  WriteString("Переполнение таблицы");
  WriteCard(переполнение,6); Write(EOL)
 END;
 Write(35C); Распечатать(T); CloseInput; CloseOutput
END XREF.
```

Теперь опишем модуль РаботаСТаблицей. Как видно из раздела описаний, он экспортирует приватный тип Таблица и операции над ним Записать и Распечатать. Обратите внимание, что структура таблиц, а значит, и алгоритмы поиска и доступа остаются скрытыми. Два наиболее вероятных способа организации таблиц – это двоичное дерево и хеш-таблица. Нами выбран первый способ. Таким образом, этот пример является дальнейшей иллюстрацией использования указателей и динамических структур данных. Модуль содержит процедуру поиска и вставки элемента в дерево, а также процедуру, которая обходит дерево и печатает содержимое таблицы (см. также раздел, посвященный динамическим структурам данных). Каждый узел дерева – запись, содержащая следующие компоненты: ключ, ссылки на левого и правого сыновей, заголовок списка, содержащего номера строк.

```
IMPLEMENTATION MODULE РаботаСТаблицей;
 FROM InOut IMPORT Write,WriteLn,WriteInt;
 FROM Storage IMPORT Allocate;

 CONST ДлинаТаблицы = 3000;
 TYPE УкДерево = POINTER TO Слово;
  УкСписок = POINTER TO Элемент;
  Элемент = RECORD номер: INTEGER;
       следующ: УкСписок
     END;
```



```
  Слово = RECORD ключ: CARDINAL; (*индекс таблицы*)
       первый: УкСписок;  (*голова списка*)
       левый,правый: УкДерево
     END;
  Таблица = УкДерево;

 VAR идент: ARRAY [0..ДлинаСлова] OF CHAR;
  ТаблИндекс: CARDINAL;
  ЛитТаб: ARRAY [0..ДлинаТаблицы-1] OF CHAR;

 PROCEDURE Инициализировать(VAR t: Таблица);
 BEGIN Allocate(t,SIZE(Слово)); t^.правый := NIL
 END Инициализировать;

 PROCEDURE Поиск(p: УкДерево): УкДерево;
  (*поиск узла с именем, равным идент*)
  TYPE Отношение = (меньше,равно,больше);
  VAR q: УкДерево;
   r: Отношение; i: CARDINAL;

  PROCEDURE сравн(k: CARDINAL): Отношение;
  (*сравнение идент с ЛитТаб[k]*)
   VAR i: CARDINAL;
    R: Отношение; x,y: CHAR;
  BEGIN i := 0; R := равно;
   LOOP x := идент[i]; y := ЛитТаб[k];
    IF CAP(x) # CAP(y) THEN EXIT END;
    IF x <= " " THEN RETURN R END;
    IF x < y THEN R := меньше
    ELSIF x > y THEN R := больше
    END;
    i := i + 1; k := k + 1
   END;
   IF CAP(x)>CAP(y) THEN RETURN больше
   ELSE RETURN меньше
   END
  END сравн;

 BEGIN q := p^.правый; r := больше;
  WHILE q # NIL DO
   p := q;
   r := сравн(p^.ключ);
   IF r = равно THEN RETURN p
    ELSIF r = меньше THEN q := p^.левый
    ELSE q := p^.правый
   END
  END;
  Allocate(t,SIZE(Слово)); (*узел не найден, вставка*)
  IF q # NIL THEN
```

```
  WITH q^ DO
   ключ := ТаблИндекс; первый := NIL;
   левый := NIL; правый := NIL
  END;
  IF r = меньше THEN p^.левый := q
  ELSE p^.правый := q
  END;
  i := 0;
  (*скопировать идентификатор в таблицу ЛитТаб*)
  WHILE идент[i] > " " DO
   IF ТаблИндекс = ДлинаТаблицы THEN
    ЛитТаб[ТаблИндекс] := " "; идент[i] := " ";
    переполнение := 1
   ELSE ЛитТаб[ТаблИндекс] := идент[i];
    ТаблИндекс := ТаблИндекс + 1; i := i + 1
   END
  END;
  ЛитТаб[ТаблИндекс] := " ";
  ТаблИндекс := ТаблИндекс + 1;
 END;
 RETURN q
END Поиск;

PROCEDURE Записать(t: Таблица;
            VAR x: ARRAY OF CHAR; n: INTEGER);
VAR  p: УкДерево; q: УкСписок; i: CARDINAL;
BEGIN i := 0;
 REPEAT идент[i] := x[i]; i := i + 1
 UNTIL (идент[i-1] = " ") OR (i = ДлинаСлова);
 p := Поиск(t);
 IF p = NIL THEN переполнение := 2
 ELSE Allocate(q,SIZE(Элемент));
  IF q = NIL THEN переполнение := 3 ELSE
   q^.номер := n; q^.следующ := p^.первый;
   p^.первый := q
  END
 END
END Записать;

PROCEDURE Распечатать(t: Таблица);

 PROCEDURE ПечЭлем(p: УкДерево);
  CONS L = 6;
   N = (ШиринаСтроки-ДлинаСлова) DIV L;
  VAR ch: CHAR;
   i,k: CARDINAL; q: УкСписок;
 BEGIN i := ДлинаСлова + 1; k := p^.ключ;
  REPEAT ch := ЛитТаб[k];
   i := i - 1; k := k + 1; Write(ch)
```



```
  UNTIL ch <= " ";
  WHILE i > 0 DO
   Write(" "); i := i - 1
  END;
  q := p^.первый; i := N;
  WHILE q # NIL DO
   IF i = 0 THEN
    WriteLn; i := ДлинаСлова + 1;
    REPEAT Write(" "); i := i - 1
    UNTIL i = 0;
    i := N
   END;
   WriteInt(q^.номер,L);
   q := q^.следующ;
   i := i - 1
  END;
  WriteLn
 END ПечЭлем;

 PROCEDURE ОбходДерева(p: УкДерево);
 BEGIN
  IF p # NIL THEN
   ОбходДерева(p^.левый);
   ПечЭлем(p);
   ОбходДерева(p^.правый);
  END
 END ОбходДерева;

BEGIN WriteLn;
 ОбходДерева(t^.правый)
END Распечатать;

BEGIN ТаблИндекс := 0; идент[ДлинаСлова] := " ";
 переполнение := 0
END РаботаСТаблицей.
```

26. ЛОКАЛЬНЫЕ МОДУЛИ

Модули, которые нам встречались до сих пор, следовало рассматривать как фрагменты текста, стоящие "бок о бок". Но модули могут быть и текстуально вложенными. Непосредственное следствие этого — то, что вложенные модули не компилируются раздельно. Они называются локальными модулями и их единственная цель — скрыть детали описания внутренних объектов.

Каждый модуль задает область видимости идентификаторов. Подразумевается, что объекты, описанные в области (модуле), видимы только в ней. Отметим, что процедуры тоже образуют область видимости. Однако здесь имеются различия.

1. Для модулей диапазон видимости объекта может быть расширен включением его идентификатора в экспортный список модуля. Тогда идентификатор становится видимым в окружающей модуль области видимости. Для процедур это невозможно.
2. Идентификатор, видимый в области, окружающей процедуру, видим также и внутри процедуры. Для модуля такое утверждение неверно, если только идентификатор не включен в импортный список этого модуля.

Правила видимости для модулей иллюстрируются следующими примерами:

```
VAR a,b: CARDINAL;
MODULE M;
 IMPORT a; EXPORT w,x;
 VAR u,v,w: CARDINAL;
 MODULE N;
  IMPORT u; EXPORT x,y;
  VAR x,y,z: CARDINAL;
   (* здесь видимы u,x,y,z *)
 END N;
 (* здесь видимы a,u,v,w,x,y *)
END M;
(* здесь видимы a,b,w,x *)
```

Если идентификатор должен пересечь несколько границ областей, то он должен быть включен ровно в столько же импортных списков (или же модуль, содержащий идентификатор, должен импортироваться целиком). Расширение области видимости из внутреннего модуля наружу достигается экспортом, снаружи вовнутрь — импортом. Правила полностью симметричны.

Рассмотрим теперь следующую структуру модулей N1, N2 и N3, вложенных в модуль M:

```
MODULE M;
 VAR a: CARDINAL;

 MODULE N1;
  EXPORT b;
  VAR b: CARDINAL;
  (* здесь видим только b *)
 END N1;

 MODULE N2;
  EXPORT c;
  VAR c: CARDINAL;
  (* здесь видим только c *)
 END N2;
```



```
 MODULE N3;
  IMPORT b,c; (* здесь видимы b и c *)
 END N3;
 (* здесь видимы a,b и c *)
END M
```

N3 импортирует идентификаторы b и c, описанные соответственно в модулях N1 и N2. Эти идентификаторы экспортированы в окружение M из локальных модулей. Если заменить модуль M "внешней средой" (в которой нельзя описывать локальные объекты), то модули N1, N2 и N3 превратятся в глобальные модули, обсуждавшиеся в предшествующем разделе (*небольшое различие имеется, поскольку экспорт из глобальных модулей может быть только квалифицируемым. Кроме того, следует иметь в виду, что у глобальных модулей существуют раздел описаний и раздел реализации. — Прим. перев.*). Фактически правила видимости для локальных и глобальных модулей совпадают. Глобальный модуль, т.е. единицу компиляции, можно назвать локальным во внешней среде.

Предположим теперь, что переменная c, экспортируемая из N2, тоже называется b. Это привело бы к коллизии имен, потому что идентификатор b уже известен в M (b экспортирован из N1). Эту проблему можно обойти, применяя квалифицируемый экспорт точно так же, как для глобальных модулей. Теперь объекты с именем b, принадлежащие N1 и N2, могут именоваться как N1.b и N2.b соответственно.

Квалифицируемый экспорт обязателен для глобальных модулей, потому что разработчик глобального модуля не знает, существует ли выбранный им идентификатор в окружающей программной среде. Для локальных модулей квалифицируемый экспорт — скорее исключение, чем правило, поскольку программист знает их окружение и, следовательно, может выбрать идентификаторы так, чтобы избежать конфликта имен.

Последнее замечание прямо касается различий модулей и процедур. И те и другие образуют некоторую область видимости (вложенную в их окружение). Но если единственной функцией модуля является создание новой области видимости, то процедура, кроме того, образует новую область существования ее локальных объектов: они исчезают при завершении процедуры. В случае модуля его локальные объекты возникают в момент создания окружающей его области существования и продолжают существовать, пока эта область не исчезнет. Однако случай модулей, локальных в процедуре, на практике встречается редко (если только не рассматривать всю программу как процедуру). Синтаксис локального модуля подобен синтаксису программного модуля:

```
$    ОписаниеМодуля =
$        "MODULE" Идентификатор [Приоритет] ";"
$        {Импорт} [Экспорт] Блок Идентификатор.
$    Приоритет = "[" КонстВыражение "]".
```

(Назначение параметра "приоритет" будет обсуждаться в разделе, посвященном параллельному исполнению.)

Следующий далее пример программы демонстрирует применение локального модуля. Цель программы – читать текст, являющийся описанием синтаксиса с помощью РБНФ, проверять правильность записи правил РБНФ и генерировать таблицу перекрестных ссылок для вводимого текста. Должны быть напечатаны две таблицы: в одной будут содержаться терминалы, т.е. строки, заключенные в кавычки, и идентификаторы, состоящие только из прописных букв, а в другой – нетерминалы, т.е. остальные идентификаторы.

Приведенная спецификация предполагает разбиение программы, подобное тому, что было у программы XREF в предшествующем разделе. Мы проведем дальнейшее разбиение задачи просмотра текста на чтение отдельных символов РБНФ, т.е. лексический анализ текста, и проверку правильности правил РБНФ, т.е. синтаксический анализ. Программа тогда будет состоять из главного модуля, называемого РБНФ, который импортирует РБНФСканер (производящий лексический анализ) и модуль РаботаСТаблицей (запоминающий и печатающий данные). Модуль РаботаСТаблицей взят без изменений из предыдущего раздела. Все три модуля, кроме того, импортируют модуль InOut.

Главная программа работает в соответствии с нисходящей стратегией разбора, подобной стратегии, которая использована в разделе, посвященном рекурсии. Разница в том, что элементами текста считаются не литеры, а символы РБНФ, получаемые по одному с помощью вызова процедуры ВзятьЛексему из модуля РБНФСканер. Кроме самой процедуры ВзятьЛексему импортируются ее результаты: переменные лекс, ид, номстр. Переменной ид присваивается строка литер, обозначающая лексический элемент, если введенный элемент – идентификатор или строка литер. Отметим, что лекс имеет тип Лексема, который также определен в модуле РБНФСканер.

```
DEFINITION MODULE РБНФСканер;
 TYPE Лексема = (идент,литерал,лкрск,лквск,лфск,
       верт,равно,точка,пкрск,пквск,пфск,другая);

 CONST ИдентДлина = 24;
 VAR лекс: Лексема; (* следующая лексема *)
  ид: ARRAY [0..ИдентДлина] OF CHAR;
  номстр: CARDINAL;

 PROCEDURE ВзятьЛексему;
 PROCEDURE ОтметитьОшибку(n: CARDINAL);
 PROCEDURE ПропускСтроки;
END РБНФСканер.
```

Этот пример еще раз иллюстрирует тот факт, что знание раздела описаний импортируемого модуля как необходимо, так и достаточно для написания импортирующего модуля.


```
MODULE РБНФ;
 FROM InOut IMPORT
  Done,EOL,OpenInput,OpenOutput,
  Read,Write,WriteLn,WriteCard,WriteString,
  CloseInput,CloseOutput;
 FROM РБНФСканер IMPORT
  Лексема,лекс,ид,номстр,ВзятьЛексему,
  ОтметитьОшибку,ПропускСтроки;
 FROM РаботаСТаблицей IMPORT ДлинаСлова,Таблица,
  переполнение,Инициализировать,Записать,Распечатать;

  (* Коды синтаксических ошибок:
     2 = ожидается ")",  6 = ожидается идентификатор
     3 = ожидается "]",  7 = ожидается "="
     4 = ожидается "}",  8 = ожидается "."
     5 = ожидается литерал,идентификатор,
                          "(","[" или "{" *)

VAR T0,T1: Таблица;
PROCEDURE пропуск(n: CARDINAL);
 (* пропустить текст до символа,
    начинающего     выражение    *)
BEGIN ОтметитьОшибку(n);
 WHILE (лекс<лкрск)OR(лекс>точка) DO ВзятьЛексему END
END пропуск;

PROCEDURE СинтВыражение;

 PROCEDURE СинТерм;

  PROCEDURE СинтМножитель;
   IF лекс = идент THEN
    Записать(T0,ид,номстр); ВзятьЛексему
   ELSIF лекс = литерал THEN
    Записать(T1,ид,номстр); ВзятьЛексему
   ELSIF лекс = лкрск THEN
    ВзятьЛексему; СинтВыражение;
    IF лекс = пкрск THEN ВзятьЛексему
    ELSE пропуск(2) END
   ELSIF лекс = лквск THEN
    ВзятьЛексему; СинтВыражение;
    IF лекс = пквск THEN ВзятьЛексему
    ELSE пропуск(3) END
   ELSIF лекс = лфск THEN
    ВзятьЛексему; СинтВыражение;
    IF лекс = пфск THEN ВзятьЛексему
    ELSE пропуск(4) END
   ELSE пропуск(5) END
  END СинтМножитель;
```

```
    BEGIN (*СинТерм*) СинтМножитель;
     WHILE лекс < верт DO СинтМножитель END
    END СинТерм;

   BEGIN (*СинтВыражение*) СинТерм;
    WHILE лекс = верт DO ВзятьЛексему; СинТерм END
   END СинтВыражение;

   PROCEDURE СинтПравило;
   BEGIN (*лекс = идент*)
    Записать(T0,ид,-INTEGER(номстр)); ВзятьЛексему;
    IF лекс = равно THEN ВзятьЛексему
    ELSE пропуск(7) END;
    СинтВыражение;
    IF лекс # точка THEN
     ОтметитьОшибку(8); ПропускСтроки END;
    ВзятьЛексему
   END СинтПравило;

  BEGIN (*Главная программа*)
   OpenInput("РБНФ");
   IF Done THEN
    OpenOutput("XREF");
    Инициализировать(T0); Инициализировать(T1);
    ВзятьЛексему;
    WHILE (лекс=идент)&(переполнение=0) DO
     СинтПравило
    END;
    IF переполнение > 0 THEN
     WriteLn; WriteString("Переполнение таблицы");
     WriteCard(переполнение,6)
    END;
    Write(35C); Распечатать(T0); Распечатать(T1);
    CloseInput; CloseOutput
   END
  END РБНФ.
```

Заслуживает внимания то, что требование раздельной печати терминалов и нетерминалов отражено в факте описания двух переменных типа Таблица. Структура программы отражает структуру синтаксиса РБНФ. Отсылаем читателя к разделу, в котором определяется РБНФ.

Задача сканера – распознавать отдельные лексические элементы, сохранять информацию о номерах строк и распечатывать вводимый текст. Дополнительная сложность возникает в связи с тем, что нужно сообщать об обнаруженных ошибках, т.е. расхождении с правилами записи синтаксиса РБНФ. В сканере хранится информация о позиции последнего прочитанного символа, и если выдается сообщение об ошибке, то вставляется строка, отмечающая место

ошибки. Имеется в виду, что входная строка должна быть прочитана вся целиком, до того как начнется ее обработка, т.е. потребуется буфер строки. Все перечисленные операции ориентированы на работу со строками текста и, следовательно, заключены в локальном модуле ОбработкаСтрок.

```
 IMPLEMENTATION MODULE РБНФСканер;
  FROM InOut IMPORT EOL,Read,Write,WriteLn,WriteCard;

  VAR ch: CHAR;

  MODULE ОбработкаСтрок;
   IMPORT EOL,ch,номстр,Read,Write,WriteLn,WriteCard;
   EXPORT ВзятьЛитеру,ОтметитьОшибку,ПропускСтроки;

   CONST ШиринаСтроки = 100;
   VAR cc: CARDINAL; (* индекс текущей литеры *)
    cc1: CARDINAL; (* число литер в текущей строке *)
    cc2: CARDINAL; (* счетчик литер в строке ошибки *)
    строка: ARRAY [0..ШиринаСтроки-1] OF CHAR;

   PROCEDURE ВзятьСтроку;
   BEGIN IF cc2 > 0 THEN
      WriteLn; cc2 := 0 (* ошибочная строка *)
    END;
    Read(ch);
    IF ch = 0C THEN (*конец файла*)
     строка[0] := 177C; cc1 := 1
    ELSE
     номстр := номстр + 1;
     WriteCard(номстр,5);
     Write(" "); cc1 := 0;
     LOOP
      Write(ch);
      строка[cc1] := ch; cc1 := cc1 + 1;
      IF (ch = EOL) OR (ch = 0C) THEN EXIT END;
      Read(ch)
     END
    END
   END ВзятьСтроку;

   PROCEDURE ВзятьЛитеру;
   BEGIN
    WHILE cc = cc1 DO
     cc := 0; ВзятьСтроку
    END;
    ch := строка[cc]; cc := cc + 1
   END ВзятьЛитеру;
```

```
  PROCEDURE ОтметитьОшибку(n: CARDINAL);
  BEGIN IF cc2 = 0 THEN
      Write("*"); cc2 := 3;
      REPEAT Write(" "); cc2 := cc2 - 1
      UNTIL cc2 = 0
    END;
   WHILE cc2 < cc DO
    Write(" "); cc2 := cc2 + 1
   END;
   Write("^"); WriteCard(n,1); cc2 := cc2 + 2
  END ОтметитьОшибку;

  PROCEDURE ПропускСтроки;
  BEGIN
   WHILE ch # EOL DO ВзятьЛитеру END;
   ВзятьЛитеру
  END ПропускСтроки;

 BEGIN cc := 0; cc1 := 0; cc2 := 0
 END ОбработкаСтрок;

 PROCEDURE ВзятьЛексему;
  VAR i: CARDINAL;
 BEGIN
  WHILE ch <= " " DO ВзятьЛитеру END;
  IF ch = "/" THEN
   ПропускСтроки;
   WHILE ch <= " " DO ВзятьЛитеру END
  END;
  IF (CAP(ch) <= "Z")&(CAP(ch) >= "A") THEN
   i := 0; лекс := литерал;
   REPEAT
    IF i < ИдентДлина THEN
     ид[i] := ch; i := i + 1
    END;
    IF ch > "Z" THEN лекс := идент END;
    ВзятьЛитеру
   UNTIL (CAP(ch) < "A") OR (CAP(ch) > "Z");
   ид[i] := " "
  ELSIF ch = "'" THEN
   i := 0; ВзятьЛитеру; лекс := литерал;
   WHILE ch # "'" DO
    IF i < ИдентДлина THEN
     ид[i] := ch; i := i + 1
    END;
    ВзятьЛитеру
   END;
   ВзятьЛитеру; ид[i] := " "
```



```
  ELSIF ch = '"' THEN
    i := 0; ВзятьЛитеру; лекс := литерал;
   WHILE ch # '"' DO
    IF i < ИдентДлина THEN
     ид[i] := ch; i := i + 1
    END;
    ВзятьЛитеру
   END;
   ВзятьЛитеру; ид[i] := " "
  ELSIF ch = "=" THEN лекс := равно; ВзятьЛитеру
  ELSIF ch = "(" THEN лекс := лкрск; ВзятьЛитеру
  ELSIF ch = ")" THEN лекс := пкрск; ВзятьЛитеру
  ELSIF ch = "[" THEN лекс := лквск; ВзятьЛитеру
  ELSIF ch = "]" THEN лекс := пквск; ВзятьЛитеру
  ELSIF ch = "{" THEN лекс := лфск;  ВзятьЛитеру
  ELSIF ch = "}" THEN лекс := пфск;  ВзятьЛитеру
  ELSIF ch = "|" THEN лекс := верт;  ВзятьЛитеру
  ELSIF ch = "." THEN лекс := точка; ВзятьЛитеру
  ELSIF ch = "177C" THEN лекс := другая; ВзятьЛитеру
  ELSE лекс := другая; ВзятьЛитеру
  END
 END ВзятьЛексему;

BEGIN номстр := 0; ch := " "
END РБНФСканер.
```

Результат работы этой программы, примененной к синтаксису языка Модула-2, приведен в приложении 1. (*В русском переводе книги в приложении 1 приведена таблица перекрестных ссылок, полученная вручную, поскольку данный вариант программы не воспринимает русские буквы. — Прим. перев.*)

27. ПОСЛЕДОВАТЕЛЬНЫЙ ВВОД И ВЫВОД

Популярность и удобство языков программирования высокого уровня достигается благодаря абстрагированию от конкретных свойств ЭВМ, на которых программа выполняется. Однако при этом сохраняются все детали алгоритма, описываемого данной программой. Операции ввода и вывода оказались тем разделом программирования, который дольше всего сопротивлялся введению абстракций. Это и неудивительно, поскольку ввод и вывод тесно связаны с работой устройств, внешних по отношению к ЭВМ. Структура, функции и работа этих устройств в большой мере зависят от их типа и марки. Многие языки программирования обычно имеют встроенные операторы для чтения и записи данных в последовательной форме без ссылки на конкретные устройства. Такая абстракция имеет много преимуществ, однако всегда существуют приложения, в которых должно быть использовано конкретное свойство какого-либо

внешнего устройства, причем стандартными операторами языка это сделать трудно или вообще невозможно. Кроме того, универсальность обычно ведет к большим накладным расходам, и операции, удобно реализуемые на одних устройствах, могут оказаться неэффективными на других. Следовательно, существует настоятельная потребность сделать свойства отдельных устройств видимыми для тех приложений, которые требуют их эффективного использования. Таким образом, упрощение и обобщение за счет опускания деталей вступает в прямое противоречие с требованием доступности деталей для эффективной реализации.

В Модуле-2 эта глубинная дилемма разрешена (или, точнее, обойдена) за счет того, что в язык вообще не включены операторы ввода и вывода. Этот крайний подход оказался возможным благодаря двум свойствам языка. Во-первых, существует такая конструкция, как модуль, позволяющая строить иерархию (библиотечных) модулей, представляющих различные уровни абстракции. Во-вторых, в языке можно выражать машинно-зависимые операции, такие, как взаимодействие с внешними устройствами. Эти операции обычно включаются лишь в модули самого низкого уровня иерархии и, следовательно, их (операции) можно отнести к так называемым средствам низкого уровня. Программа, не использующая детали работы с конкретными устройствами, импортирует процедуры из стандартных модулей, находящихся на высоких уровнях упомянутой иерархии. Программа же, требующая высокой эффективности или использующая специфические свойства конкретного устройства, может применить либо модули низкого уровня, так называемые драйверы устройств, либо непосредственно примитивы языка. В последнем случае получится немобильная программа, так как будет происходить обращение к особенностям конкретной ЭВМ или операционной системы.

В этом контексте невозможно привести примеры операций над внешними устройствами, описываемых на низком уровне иерархии модулей, поскольку существует большое разнообразие таких устройств. Мы ограничим дальнейшее изложение, приведя лишь типичную иерархию модулей, используемых при выполнении операций стандартного ввода и вывода и описав стандартный модуль InOut, уже встречавшийся в примерах предыдущих разделов. Кроме того, назовем некоторые операции, которые должны быть в любой реализации Модулы. Однако мы не хотим четко определять ни имена модулей, содержащих эти операции, ни набор остальных операций, включаемых в такие модули. Подчеркнем еще раз, что иерархия таких модулей и их экспорт не относятся собственно к языку, хотя следует отдавать себе отчет, что без наличия таких модулей программирование оказалось бы слишком обременительным.

В общем случае будем разделять ввод-вывод на видимый и невидимый. Видимый ввод и вывод служат для связи между ЭВМ и пользователем. В основном его элементы – литеры, имеющие тип CHAR; исключение составляет графический ввод-вывод. Видимые данные – ввод через клавиатуру, считыватель с перфокарт и т.д.

Видимый вывод генерируется дисплеями, печатающими устройствами. Невидимый ввод-вывод или обмен осуществляется между ЭВМ и так называемыми внешними запоминающими устройствами, такими, как магнитные диски и ленты. Невидимый ввод может осуществляться с датчиков (как, например, в лабораториях), а вывод – на устройства, управляемые ЭВМ, такие, как графопостроители, заводские сборочные линии, светофоры, сети передачи данных. Данные для невидимого ввода-вывода могут иметь любой тип, а не только тип CHAR.

Подавляющее большинство операций ввода-вывода и видимого, и невидимого можно рассматривать как последовательные. Данные для этих операций имеют тип, который не принадлежит к основным структурным типам Модулы, таким, как массив и запись. Но тем не менее мы дали ему имя. Такой тип называется потоком. Для него характерны следующие особенности:

1. Все элементы потока имеют один и тот же тип, базовый тип потока. Если этот тип – CHAR, то поток называется текстовым потоком.

2. Число элементов потока заранее неизвестно. Следовательно, поток – динамическая структура (простейший случай). Число элементов называется длиной потока, и поток с нулевым числом элементов называется пустым потоком.

3. Поток может модифицироваться только добавлением элементов в конец (или удалением потока целиком). Добавление элемента называется записью (не путать с типом запись).

4. В каждый момент времени виден (доступен) только один элемент, а именно элемент, находящийся в текущей позиции потока. Доступ к этому элементу называется чтением. Операция чтения обычно продвигает позицию потока к следующему элементу.

5. Каждый поток имеет режим: поток может либо читаться, либо писаться. Следовательно, каждый поток характеризуется состоянием, содержащим длину, позицию и режим.

Заметим, между прочим, что поток в том виде, как описано выше, возможно, наиболее удачный пример абстракции данных. Несомненно, он используется в повседневной практике гораздо шире, чем часто приводимые примеры стеков и очередей. Язык Паскаль включает его в набор своих основных структурных типов наряду с массивами, записями и множествами. В Паскале потоки называются последовательными файлами, а видимые потоки (с базовым типом CHAR) – текстовыми файлами.

До того как мы начнем рассмотрение иерархии модулей, осуществляющих ввод и вывод, хотелось бы отметить наличие двух различных типов операций ввода-вывода. В особенности это касается видимого ввода и вывода. С одной стороны, происходит реальная передача данных между ЭВМ и внешними устройствами. Сюда

же входят такие операции, как запуск и проверка состояния внешних устройств: клавиатуры, дисплея, печатающего устройства. С другой стороны, необходимы операции преобразования данных из одной формы в другую. Если, например, значение выражения типа **CARDINAL** нужно выдать на дисплей, то внутреннее представление необходимо преобразовать в литерное в виде последовательности десятичных цифр. Затем в дисплее происходит перевод литерного представления (обычно содержащего 8 битов на литеру) в матрицу видимых точек или линий. Перевод внутреннего представления числа в последовательность литер можно считать машинно-независимой операцией, и, следовательно, это очевидный кандидат на отделение от операций, характерных для устройства. Эта операция может производиться одной и той же процедурой, независимо от того, будет ли поток запоминаться на диске или высвечиваться на экране дисплея. Такое преобразование называется форматированием.

Третий класс функций, которые могут быть легко выделены, относится к устройствам, связанным более чем с одним потоком, в основном это устройства дисковой памяти. Мы обращаемся в этом случае к операциям выделения дисковой памяти и связывания имен с отдельными потоками или файлами. Учитывая тот факт, что потоки (и файлы) – динамические структуры, приходим к выводу, что операции выделения памяти весьма сложны. Именование отдельных файлов, и в особенности работа с директориями для быстрого поиска отдельного файла, – вторая задача, требующая тщательно разработанного механизма. И выделение памяти, и работа с директориями – задачи, относящиеся к резидентной части операционной системы. Похоже, что существует столько же способов решения этих задач, сколько и самих операционных систем. И ввод-вывод – именно та область, где уровни абстракции операций существенно различаются в разных операционных системах, чрезвычайно затрудняя выработку обязательных соглашений о примитивах работы с файлами, которые бы были и независимы от операционной системы, и эффективно реализуемы на многих (или хотя бы больше чем на одной) операционных системах. По этой причине наше решение заключается в том, чтобы предложить иерархию модулей, оставив выбор уровня подключения к этой иерархии на усмотрение программиста. Подключение на высоком уровне обеспечивает простоту понятий и мобильность программ (возможно, за счет эффективности); низкий уровень подключения к иерархии открывает перед программистом весь диапазон возможностей, предоставляемых операционной системой. Эти преимущества достигаются, однако, за счет меньшей мобильности программ. В последнем случае программисту настоятельно рекомендуется размещать системно-зависимые операторы в минимально возможном числе мест программы. Вершина нашей иерархии модулей образуется модулем **InOut**. Он содержит два текстовых потока, один из которых – стандартный входной источник, а другой – стандартный выходной поток. Модуль InOut предлагает следующие возможности:

1. Набор процедур чтения данных из входного потока in предназначен для ввода форматированных данных. Это процедуры

```
Read(ch)
ReadString(s)
ReadInt(x)
ReadCard(x)
```

Конец потока определяется проверкой экспортируемой переменной **Done**. Ее значение равно **FALSE**, если операция чтения оказалась безуспешной из-за того, что встретился конец потока. В этом случае процедура **Read(ch)** присваивает переменной **ch** значение **0C**. Следовательно, типичная схема программы последовательного ввода имеет вид

```
Read(ch);
WHILE Done DO обработать(ch); Read(ch);
END
```

2. Набор процедур записи в поток out служит для вывода форматированных данных. Это следующие процедуры:

```
Write(ch)
WriteString(s)
WriteLn
WriteInt(x,n)
WriteCard(x,n)
WriteOct(x,n)
WriteHex(x,n)
```

3. Процедуры

```
OpenInput(s)
OpenOutput(s)
CloseInput
CloseOutput
```

предназначены для связи файлов со стандартными потоками **in** и **out**. Если не вызывалась программа **OpenInput**, то предполагается, что входные данные поступают со стандартного входного устройства, обычно с клавиатуры оператора. Вызов **OpenInput** приводит к запросу имени файла со стандартного входного устройства и привязке потока **in** к указанному файлу. Аналогично (до вызова **OpenOutput**), выходные данные направляются на стандартное выходное устройство, обычно операторский терминал, а после вызова происходит их привязка к указанному файлу. Вызов процедуры **CloseInput** (**CloseOutput**) возвращает ввод (вывод) на стандартное устройство. Открытые файлы перед завершением программы должны быть закрыты.

В модуле InOut эффективно достигнута независимость от используемой операционной системы посредством такой абстракции, как поток, и за счет упрятывания двух стандартных потоков, описания которых могут включать характеристики операционной системы. Эти характеристики могут потребоваться для достижения эффективности в модулях низкого уровня. Модуль также скрывает такие системно-зависимые средства, как способ именования файлов, операции их открытия и закрытия. Кроме того, одни и те же процедуры форматирования применяются как при работе с клавиатурой и дисплеем, так и при работе с файлами. Дальнейшие подробности можно получить из определения модуля InOut, приведенного в приложении.

Форматный ввод и вывод действительных чисел осуществляется стандартным сопутствующим модулем <u>RealInOut</u>. Он предоставляет процедуры

```
ReadReal(x)
WriteReal(x,n)
```

и имеет доступ к потокам через посредство процедур Read и Write модуля InOut. Следовательно, перенаправление ввода и вывода вызовами OpenInput и OpenOutput влияет также и на процедуры из RealInOut, определение которого также приведено в приложении.

Модуль InOut содержит переключатель, направляющий потоки данных либо на терминал, либо в файловую систему в зависимости от того, был открыт файл или нет. На уровне, лежащем ниже чем InOut, мы, следовательно, обнаружим два модуля: один для ввода с терминала и вывода на терминал, и другой модуль, представляющий файловую систему. Здесь будут описаны оба эти модуля, поскольку во многих случаях программист пожелает иметь доступ к ним непосредственно.

Для терминального ввода-вывода мы определяем модуль <u>Terminal</u>. Он экспортирует процедуру Read для чтения данных с клавиатуры и процедуры <u>Write</u> и <u>WriteString</u> для записи на экран или на печатающее устройство терминала (см. приложение 2).

Другой модуль, тоже лежащий уровнем ниже в иерархии и связывающий InOut с <u>файловой системой</u> конкретной ЭВМ, включает в себя понятие потока. Учитывая тот факт, что этот модуль должен быть близок к конкретной файловой системе, мы не даем его точного определения, не желая накладывать ограничения на возможные реализации. В различных окружениях может меняться даже его имя. Тем не менее перечисляется совокупность описаний, которые программист может считать присутствующими во всех Модула-системах. Эти описания будут перечислены ниже, но вначале сосредоточим внимание на потоках, т.е. на структурах последовательного доступа.

Мы будем различать два вида потоков, а именно <u>текстовый поток</u> с базовым типом CHAR и <u>поток слов</u> с базовым типом WORD. Тип WORD зависит от марки ЭВМ, но он всегда должен быть совместим при передаче параметров со всеми типами, занимающими в памяти одно слово, такими, как INTEGER, CARDINAL и BITSET. (Более подробно этот тип будет описан в разделе, посвященном средствам программирования низкого уровня.)

Описываемый модуль экспортирует тип, обозначающий поток. Здесь мы будем обозначать этот тип именем STREAM (поток), однако возможны и другие имена, поскольку в некоторых реализациях этот тип может экспортироваться из файловой системы. Такой экспорт в действительности более предпочтителен, поскольку в конечном счете поток реализуется именно в виде файла. Файл превращается в поток простым ограничением набора применимых к нему операций только операциями последовательного ввода и вывода. Это следующие операции (если s имеет тип STREAM):

```
ReadChar(s,ch)      ReadWord(s,w)
WriteChar(s,ch)     WriteWord(s,w)
```

Модуль также экспортирует средства для установки и проверки состояния потока, в частности, того, достигнут при вводе конец потока или нет. Если модуль, содержащий поток, не включен непосредственно в файловую систему, то он экспортирует процедуры для связи нового потока с файлом (из файловой системы) и для отсоединения потока от файла, когда поток больше не требуется.

В качестве такого модуля приведем раздел описаний модуля, реализованного в операционной системе RT-11 на мини-ЭВМ PDP-11. Этот модуль, называемый <u>Streams</u>, импортирует тип FILE из модуля <u>Files</u>. Фактически этот модуль определяет взаимодействие программ на Модуле с операционной системой RT-11, которая идентифицирует файлы по порядковым номерам, так называемым номерам каналов. Процедура Connect связывает файл системы RT-11, т.е. объект типа FILE, с потоком, т.е. объектом типа STREAM, и определяет, будет ли это символьный поток или поток слов.

```
DEFINITION MODULE Streams; (* для RT-11 *)
 FROM SYSTEM IMPORT WORD;
 FROM Files IMPORT FILE;

 TYPE STREAM;

 PROCEDURE Connect(VAR s: STREAM; f: FILE;
                   ws: BOOLEAN);
  (* связать поток s с открытым файлом f.
     В RT-11 f - номер канала.
     ws = "s - поток слов, а не литер" *)
 PROCEDURE Disconnect(VAR s: STREAM;
                      закрытьфайл: BOOLEAN);
 PROCEDURE WriteWord(s: STREAM; w:WORD);
 PROCEDURE WriteChar(s: STREAM; ch: CHAR);
 PROCEDURE EndWrite(s: STREAM);
```

```
 PROCEDURE ReadWord(s: STREAM; VAR w:WORD);
 PROCEDURE ReadChar(s: STREAM; VAR ch: CHAR);
 PROCEDURE EOS(s: STREAM): BOOLEAN;
 PROCEDURE Reset(s: STREAM);
 PROCEDURE SetPos(s: STREAM; ст,мл: CARDINAL);
 PROCEDURE GetPos(s: STREAM; VAR ст,мл: CARDINAL);
END Streams.
```

В случае текстовых потоков процедуры **ReadChar** и **WriteChar** осуществляют необходимые преобразования представления концов строк. В Модуле строка завершается одной литерой EOL, а в файле RT-11 – парой литер – cr и lf (15C,12C). Если достигнут конец потока, то процедура ReadChar(s,ch) присваивает переменной ch значение 0C.

Программист, желающий использовать другие файловые операции операционной системы, должен получить к ним непосредственный доступ, импортировав их из модуля Files. Для удобства программиста раздел определений модуля Files приведен ниже. Особый интерес представляют процедуры LookUp (найти), Create (создать) и Close (закрыть). Они требуются во всех программах, использующих модуль Streams, поскольку файл должен быть либо найден в директории RT-11, либо создан до привязки к нему потока. Отметим, что имя файла в RT-11 состоит ровно из 12 литер: первые три обозначают устройство, следующие шесть – собственно имя файла и три последних образуют так называемое <u>расширение имени</u>.

```
DEFINITION MODULE Files;  (* Ch. Jacobi для RT-11 *)
 FROM SYSTEM IMPORT ADDRESS,WORD;
 TYPE FILE = [0..15];
   FileName = ARRAY [0..11] OF CHAR; (* имя файла *)

 PROCEDURE Lookup(f: FILE; fn: FileName; VAR ответ: INTEGER);
  (* поиск файла f в директории
     ответ: >= 0 = все в порядке, длина файла
            <  0 = ошибка: -1 = канал занят,
                           -2 = файл не найден *)

 PROCEDURE Create(f: FILE; fn: FileName; VAR ответ: INTEGER);
  (* создать новый файл f, не занося его в директорию.
     ответ: >= 0 = все в порядке, длина файла
            <  0 = ошибка: -1 = канал занят,
                           -2 = нет места *)

 PROCEDURE Delete(f:FILE; fn: FileName; VAR ответ: INTEGER);
  (* удалить файл f и его имя из директории
     ответ: >= 0 = все в порядке, длина файла
            <  0 = ошибка: -1 = канал занят,
                           -2 = файл не найден *)
```



```
 PROCEDURE Close(f: FILE);
  (* закрыть файл f и занести его в директорию *)

 PROCEDURE Release(f: FILE);
  (* закрыть файл f, не занося его в директорию *)

 PROCEDURE ReadBlock(f: FILE; p: ADDRESS;
       номбл,счслов: CARDINAL; VAR ответ: INTEGER);
  (* чтение из файла f
     p: адрес буфера
     номбл: номер первого читаемого блока
     счслов: сколько слов нужно прочесть
     ответ: >= 0 = число пересланных слов
            < 0 = ошибка: -1 = серьезная ошибка,
                          -2 = канал не открыт  *)

 PROCEDURE WriteBlock(f: FILE; p: ADDRESS;
       номбл,счслов: CARDINAL; VAR ответ: INTEGER);

  (* запись в файл f
     p: адрес буфера
     номбл: с какого блока начнется запись
     счслов: сколько слов нужно записать
     ответ: >= 0 = число пересланных слов
            < 0 = ошибка: -1 = серьезная ошибка,
                          -2 = канал не открыт  *)

 PROCEDURE Rename(f: FILE; new,old: FileName;
                  VAR ответ: INTEGER);

  (* переименовывает файл f; файл должен быть закрыт.
     ответ: 0 = все в порядке
          < 0 = ошибка: -1 = канал занят,
                        -2 = файл не найден  *)

END Files.
```

Мы завершим описание работы с файлами в системе RT-11 тем, что укажем иерархию модулей.

InOut —> Streams —> Files —> RT-11.

Следовательно, вызов, например, процедуры **Read** модуля **InOut** подразумевает вызов процедуры **ReadChar** модуля **Streams**, которая может вызывать процедуру ReadBlock модуля Files, что в свою очередь подразумевает вызов системного примитива для чтения сектора диска. В качестве примера приведем последовательную обработку файла DATA.IN, рассматриваемого как поток слов, и запись результатов в файл DATA.OUT в виде потока символов.

```
FROM Files IMPORT FILE,Lookup,Create,Close;
FROM Streams IMPORT
 STREAM,Connect,ReadWord,WriteChar,EOS,Disconnect;

VAR f1,f2: FILE;
 s1,s2: STREAM;
 x: CARDINAL; y: CHAR; ответ: INTEGER;

BEGIN f1 := 1; f2 := 2; (* номера каналов в RT-11 *)
 Lookup(f1,"DK DATA  IN ",ответ);
 Create(f2,"DK DATA  OUT",ответ);
 Connect(s1,f1,TRUE); Connect(s2,f2,FALSE);
 ReadWord(s1,x);
 WHILE NOT EOS(s1) DO
  обработать(x,y); WriteChar(s2,y); ReadWord(s1,x)
 END;
 Disconnect(s1,FALSE); Disconnect(s2,TRUE)
END
```

Примером реализации, в которой уровень модуля Streams совпадает с Files, т.е. реализации, представляющей собой файловую систему, включающую понятие потока, является система Medos для ЭВМ Лилит (Lilith). Фрагмент программы, решающей все ту же задачу последовательной обработки файла, которая уже рассматривалась ранее, приведен ниже. Упрощение, полученное за счет ликвидации промежуточного модуля (Streams), очевидно.

```
FROM FileSystem IMPORT
 File,Lookup,ReadWord,WriteChar,Close;

VAR f1,f2: File;
 x: CARDINAL; y: CHAR;

BEGIN
 Lookup(f1,"DK.DATA.IN",FALSE);
 Lookup(f2,"DK.DATA.OUT",TRUE);
 ReadWord(f1,x);
 WHILE NOT f1.eof DO
  обработать(x,y); WriteChar(f2,y); ReadWord(f1,x)
 END;
 Close(f1); Close(f2)
END
```

Аккуратный программист, несомненно, вставит проверки успешного поиска файла. В таких деталях различные реализации, как и следовало ожидать, отличаются друг от друга. Например, в RT-11 успешность поиска определяется анализом параметра reply (ответ) процедуры LookUp, а в системе Medos – анализом компоненты res файловой переменной (которая имеет структуру записи). Более конкретно, проверка имеет вид: "f1.res = done", где done (сделано) – это константа перечислимого типа Response (ответ), экспортируемая из модуля FileSystem. Внимательный программист должен также иметь в виду, что существует еще одно небольшое отличие между двумя версиями: в то время как процедура Create (для RT-11) всегда открывает новый файл, процедура LookUp (в Medos) создает новый файл, только если третий параметр – TRUE и не существует уже файла с заданным именем. Поле f2.new позволяет определить, действительно ли файл f2 новый. Другими словами, старый файл может быть перезаписан.

28. ЭКРАННЫЙ ВВОД И ВЫВОД

При последовательном вводе и выводе подразумевается, что элементы данных могут передаваться без явного указания позиции. Это естественно, если позиция неявно определяется запоминающим устройством, таким, как лента (которая по определению образует последовательность) или клавиатура (с которой данные поступают в четкой временной последовательности) или печатающее устройство (где позиции литер определяются механическим движением устройства). Даже если запоминающее устройство и допускало бы большую гибкость, все равно последовательный вывод удобен, если структура данных существенно последовательна. Например, текст – существенно последовательная структура, и отсутствие необходимости хранить информацию о позиции каждой литеры сильно упрощает задачу чтения и записи. И наконец, последовательный ввод и вывод весьма экономен, поскольку легко реализуется буферизация данных между процессами (устройствами), работающими параллельно. Все это объясняет, почему последовательная работа с данными так широко распространена и желательна.

Однако существуют приложения, требующие рассмотрения данных не в виде последовательности. Для таких приложений характерно, что каждый элемент несет информацию о своей позиции. Например, дисковая память допускает избирательное чтение и запись отдельных блоков данных, так называемых секторов. Часто большие наборы данных пишутся последовательно, а читаются выборочно. (Эти возможности были отмечены в модулях Files и FileSystem из предыдущего раздела.)

С недавних пор важность вывода, не являющегося последовательным, возросла из-за распространения визуальных средств вывода, т.е. дисплеев, высвечивающих данные на экране. Большинство дисплейных терминалов до сих пор работает в последовательном режиме лишь потому, что данные – большей частью просто текст, а также и потому, что последовательный способ обработки привычен и многие пользователи просто не представляют себе преимуществ непоследовательной обработки.

Видится два основных мотива использования вывода, отличного от последовательного.

1. Данные включают элементы, не имеющие последовательной природы, такие, как линии, таблицы, схемы или рисунки, т.е. так называемую графику.
2. Экран должен использоваться для вывода нескольких последовательностей выходных данных независимо и параллельно, т.е. экран используется для имитации нескольких дисплеев, причем каждый из них несет информацию о своей позиции относительно всего экрана.

Далее мы опишем два модуля, которые предоставляют названные возможности. Поскольку непоследовательные операции обеспечивают существенно большую гибкость, они используются в очень широком диапазоне приложений. Поэтому эта область гораздо труднее для стандартизации. Таким образом, модули, приводимые далее, следует рассматривать лишь как предварительные предложения, а не как "окончательные решения". Но тем не менее они оказались очень полезными и удобными во многих практических приложениях.

Важное подмножество универсальных графических средств – вычерчивание прямых. Если к этому добавить возможность вывода текста (короткие строки для заголовков и т.д.), то вычерчивание прямых может оказаться во многих случаях, таких, как вывод таблиц и диаграмм, вполне достаточным. Для этих приложений мы предлагаем модуль LineDrawing (вычерчивание прямых). В этом модуле предполагается наличие прямоугольной области экрана. Из модуля экспортируется ширина (width) и высота (height) экрана, заданные в единицах горизонтальных и вертикальных координат соответственно. Экран считается матрицей точек, называемой растром. Возможен доступ к любой точке (элементу растра); здесь мы предполагаем, что любая из них может быть закрашена в черный или белый цвет. Распространение этого понятия на многозначные точки, описывающие либо тона серого цвета разной яркости, либо различные цвета, с концептуальной точки зрения очевидно.

Наиболее важные процедуры, содержащиеся в модуле LineDrawing, называются **dot** (точка) и **area** (область). Вызов

```
dot(c,x,y)
```

закрашивает элемент растра с координатами x,y, шириной и высотой w и h соответственно в "цвет" c, где c = 0 будет обозначать белый цвет, а c = 1 – черный. Конечно, x и y должны лежать внутри области экрана, т.е. 0 <= x < width, 0 <= y < height. Мы предполагаем, что координаты 0,0 обозначают левый нижний угол прямоугольной области экрана. Вызов

```
area(c,x,y,w,h)
```

закрашивает прямоугольник с координатами левого нижнего угла x,y в "цвет" c. Можно считать, что c = 0 обозначает белый, c = 1 – светло-серый, c = 2 – темно-серый и c = 3 – черный цвет; однако другие реализации могут предлагать более широкий набор значений или даже настоящие цвета. Способ представления диапазона серого тоже не фиксирован. Например, значение c может непосредственно управлять интенсивностью электронного пучка либо же использоваться для выбора комбинации точек, размножаемой в задаваемом прямоугольнике. Использование процедуры **area** продемонстрировано в модуле Ферзи, приведенном в разд. 14.


```
DEFINITION MODULE LineDrawing;
  TYPE PaintMode = (replace,add,invert,erase);

  VAR Px,Py: INTEGER; (* текущие координаты пера *)
    mode: PaintMode;
          (* текущий режим рисования и копирования *)
    width: INTEGER; (* ширина картинки, только чтение *)
    height: INTEGER; (* высота картинки, только чтение *)
    CharWidth: INTEGER; (* ширина литеры *)
    CharHeight: INTEGER; (* высота литеры *)

  PROCEDURE dot(c: CARDINAL; x,y: INTEGER);
   (* поставить точку с координатами x,y *)

  PROCEDURE line(d,n: CARDINAL);
   (* нарисовать прямую длины n
      в направлении d (угол = 45*d градусов) *)

  PROCEDURE area(c: CARDINAL; x,y,w,h: INTEGER);
   (* закрасить прямоугольную область с координатами левого
      нижнего угла x,y, шириной w и высотой h в цвет c
      0=белый,1=светло-серый,2=темно-серый,3=черный *)

  PROCEDURE copyArea(sx,sy,dx,dy,dw,dh: INTEGER);
   (* скопировать прямоугольную область sx,sy,dw,dh
      в область dx,dy,dw,dh *)

  PROCEDURE clear; (* очистить экран *)

  PROCEDURE Write(ch: CHAR);
   (* поставить символ ch в позиции пера *)

  PROCEDURE WriteString(s: ARRAY OF CHAR);

END LineDrawing.
```

Процедура **line** показывает, что даже в графике последовательный режим иногда бывает весьма желательным из-за удобства работы с ним. При использовании этой процедуры мы предполагаем существование некоторого воображаемого пера,

которое чертит прямые линии. Это перо имеет перед вызовом процедуры некоторую подразумеваемую позицию. Вызов

```
line(d,n)
```

передвигает его в направлении d на n единиц растра. Таким образом, последовательность таких вызовов прочертит ломаную линию и при этом не потребуется каждый раз непосредственно указывать положение пера. В предлагаемой версии процедуры мы допускаем лишь несколько выделенных направлений, а именно лишь прямые с углами вида d*45^0, где d = 0,1,...,7, причем d = 0 означает движение в положительном направлении по оси x, т.е. вправо. Такой режим вычерчивания прямых иногда называется черепашьей графикой.

Положение пера задается экспортируемыми переменными Px и Py. Они используются при установке начала первого сегмента ломаной и при переустановке пера для вычерчивания других ломаных. Использование черепашьей графики из модуля LineDrawing демонстрируется следующим примером программы. Эта программа рисует кривую, изобретенную математиком Серпински, которая заполняет всю плоскость. Эта программа также представляет собой изящный пример использования рекурсивных процедур для построения рекурсивно определенного изображения.

```
MODULE Серпински;
 FROM Terminal IMPORT Read;
 FROM LineDrawing IMPORT widtn,height,Px,Py,clear,line;

 CONST РазмерКвадрата = 512;

 VAR i,h,x0,y0: CARDINAL; ch: CHAR;

 PROCEDURE A(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   A(k-1); line(7,h); B(k-1); line(0,2*h);
   D(k-1); line(1,h); A(k-1)
  END
 END A;

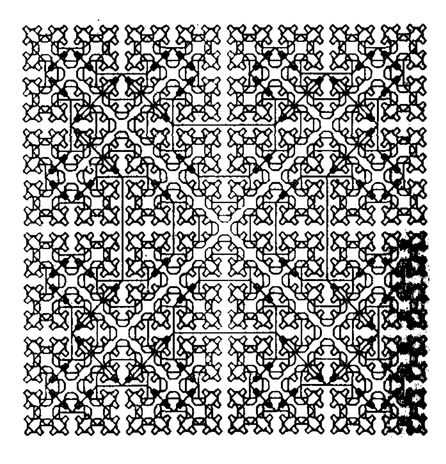
 PROCEDURE B(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   B(k-1); line(5,h); C(k-1); line(6,2*h);
   A(k-1); line(7,h); B(k-1)
  END
 END B;
```



```
 PROCEDURE C(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   C(k-1); line(3,h); D(k-1); line(4,2*h);
   B(k-1); line(5,h); C(k-1)
  END
 END C;

 PROCEDURE D(k: CARDINAL);
 BEGIN
  IF k > 0 THEN
   D(k-1); line(1,h); A(k-1); line(2,2*h);
   C(k-1); line(3,h); D(k-1)
  END
 END D;

BEGIN clear; i := 0; h := РазмерКвадрата DIV 4;
 x0 := CARDINAL(width) DIV 2;
 y0 := CARDINAL(height) DIV 2 + h;
 REPEAT i := i + 1; x0 := x0 - h;
  h := h DIV 2; y0 := y0 + h; Px := x0; Py := y0;
  A(i); line(7,h); B(i); line(5,h);
  C(i); line(3,h); D(i); line(1,h); Read(ch)
 UNTIL (i = 6) OR (ch = 33C);
 clear
END Серпински.
```



Модуль **LineDrawing** содержит процедуры для вычерчивания прямых линий и прямоугольников. Более сложные конфигурации можно получить закрашиванием отдельных точек с помощью процедуры **dot**. Окружность является довольно важным геометрическим элементом. Поэтому мы опишем здесь процедуру, рисующую окружности с координатами центра x,y и радиусом r. Любопытная особенность этого метода состоит в том, что он не использует чисел типа REAL, а потому сравнительно эффективен.

Уравнение окружности имеет вид x^2 + y^2 = r^2. После того как нарисована точка в позиции x,y, мы хотим вычислить координаты следующей точки, а именно x+dx и y+dy. Отношение dy/dx можно найти, продифференцировав уравнение кривой: получим dy/dx = -x/y. Следовательно, мы можем задать dx = -k*y и dy = k*x, *где k – достаточно малая константа, определяющая* степень грубости изображения окружности, аппроксимируемой последовательностью сторон многоугольника. Будем использовать лишь дробные числа с фиксированной точкой, представленные в виде *масштабированных целых чисел. Возьмем величину c1 = 1/k* и вычислим очередные значения x и y по формулам

```
x := x - y DIV c1; y := y + x DIV c1
```

Если предположить, что в нашем распоряжении имеется экран, имеющий ширину и высоту не более 512 точек, то для представления координат потребуется 9 бит. Предполагая далее, что длина машинного слова составляет 16 бит, получаем, что 7 бит остается для дробной части. Таким образом, можно считать, что наши целые координаты имеют на самом деле двоичную точку, сдвинутую на 7 разрядов влево. Усеченная целая часть координаты x получается делением ее на c2 = 2^7 = 200B. Константа c1 выбирается настолько большой, чтобы при максимальных значениях x и y вычисляемые приращения были равны единице.

```
PROCEDURE Окружность(x0,y0,r: INTEGER);
 CONST c1 = 400B; c2 = 200B;
 VAR x,y: INTEGER;
BEGIN
 r := r*c2; x := r; y := 0; r := r - 1;
 REPEAT dot(1,x DIV c2 + x0,y DIV c2 + y0);
  x := x - y DIV c1; y := y + x DIV c1
 UNTIL (x >= r) & (y <= 0)
END Окружность
```

Примечание: Этот алгоритм в действительности более хитроумен, чем может показаться, и для доказательства *правильности его функционирования* требуется тщательный численный анализ эффектов округления.

Средства графического вывода становятся значительно полезнее, если их объединить с подходящим устройством ввода. Допустим, что имеется некоторое устройство, позволяющее регистрировать перемещения по плоскости. С его помощью можно вводить позицию в терминах координат x и y. Более того, мы предполагаем, что с этим устройством связано несколько кнопок, состояние которых может быть считано.

Мы называем это устройство ввода мышью. Название связано с конкретным оформлением в виде устройства, перемещаемого рукой по столу. У него три кнопки (глаза), и оно подсоединяется к клавиатуре тонким, возможно серым, кабелем. Все это и определило его название.

Положение мыши отображается на экран с помощью отметки, называемой курсором. Тем самым *осуществляется привязка* устройства к экрану и к положению отдельных объектов, изображенных на экране. Способ изображения курсора и связь его положения на экране с перемещениями мыши по столу скрыты внутри модуля, который будет назван **Mouse** (Мышь). Пользователю не нужно знать ни его деталей, ни даже того, реализован ли модуль с использованием аппаратных средств или же только программными средствами. Однако существенно то, что курсор может быть установлен на любой элемент растра. Тем самым устройство ввода получает наибольшее полезное разрешение.

Процедуры, используемые для управления мышью, – **TrackMouse** (СледМыши) и **FlipCursor** (ПереключитьКурсор). **TrackMouse** используется для отслеживания перемещений мыши, т.е. руки пользователя. Она считывает положение мыши, изображает в соответствующем положении курсор и присваивает его координаты экспортируемым переменным Mx и My. Обычно эта процедура вызывается внутри короткого цикла, выполняемого, пока не будет нажата одна из кнопок, что сигнализирует о необходимости выполнить некоторое действие, связанное с указанными координатами. Процедура **FlipCursor** переключает внутренний флажок, включающий и выключающий изображение курсора на экране. Эта процедура важна, если необходимо иногда на время удалять курсор с экрана. Например, модуль **Mouse**, реализованный автором на машине Лилит, требует, чтобы курсор удалялся с экрана во время рисования, записи или стирания элементов изображения. Текущее значение флажка представляется экспортируемой переменной **curOn** (курсор включен).

Процедура **ShowMenu** (ПоказМеню) предоставляет удобный способ ввода команд. Ее вызов приводит к высвечиванию в текущей позиции курсора на экране списка командных слов, задаваемых текстовым параметром. Выдаваемая таким образом информация должна рассматриваться как меню команд, доступных в текущий момент. *Последующими перемещениями мыши* можно выбрать одну из команд. Предполагается, что **ShowMenu** должна вызываться при нажатии специальной клавиши мыши. *При отпускании клавиши управление* возвращается со значением параметра, указывающим выбранное

командное слово. Использование таких меню – удобное средство гибкого ввода команд с использованием единого простого устройства ввода с одновременным указанием позиции объекта, к которому применяется команда. Меню команд появляется в текущей позиции курсора, т.е. именно там, где сфокусировано внимание работающего.

```
DEFINITION MODULE Mouse;

 VAR keys: BITSET; (* кнопки мыши *)
  Mx,My: INTEGER;  (* координаты курсора и мыши *)
  curOn: BOOLEAN;  (* флажок состояния курсора:
                      начальное значение = FALSE *)
  mode: CARDINAL;

 PROCEDURE TrackMouse;
  (* прочитать координаты мыши Mx,My и состояние кнопок;
     сдвинуть курсор в соответствующую позицию *)

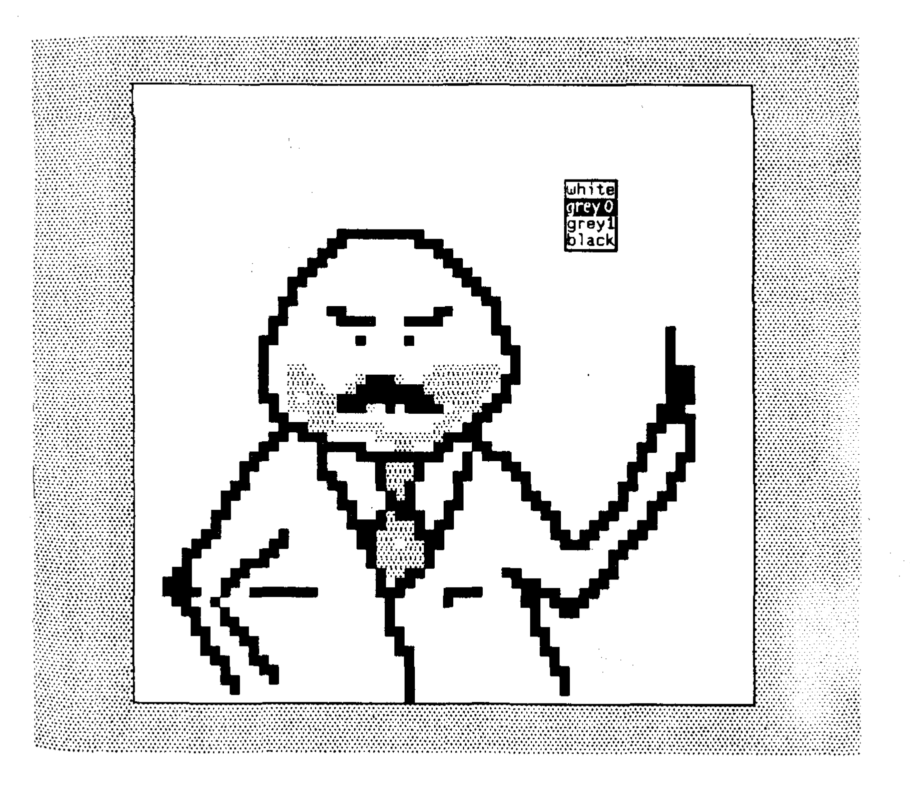
 PROCEDURE FlipCursor;
  (* переключить состояние флажка курсора *)

 PROCEDURE ShowMenu(текст: ARRAY OF CHAR;
                    VAR Выбор: INTEGER);
  (* высвечивает текст меню в текущей позиции курсора;
     последующими движениями мыши выбирается нужная
     команда. Выбор происходит при отпускании кнопки.
     Выбор = 0 означает, что никакая команда не была
     выбрана. В строке "текст" командные слова
     разделяются литерой "|". Должно быть не более 8
     команд, и командные слова не должны содержать
     более 7 литер *)
END Mouse.
```

Использование модуля LineDrawing совместно с модулем Mouse демонстрируется следующей программой Рисование. Она позволяет рисовать картинки на квадратном участке, содержащем 64*64 растровых "элемента". Каждый из этих "элементов" представляется квадратиком 8*8 растровых элементов (пикселов). Используя меню, можно выбирать для рисования различные цвета или яркости. Эту программу можно дополнить и усовершенствовать многими способами, но основная ее структура такова:

```
инициализировать экран;
FlipCursor; (*включить*)
REPEAT TrackCursor;
  IF (нажата кнопка) & (мышь передвинута) THEN
   FlipCursor; (*выключить*)
   выполнить нужное действие;
   FlipCursor; (*включить*)
  END;
  BusyRead(ch);
  (* проверка нажатой клавиши *)
UNTIL ch = ESC;
очистить экран
```


Этот пример также демонстрирует то, каким образом можно "одновременно" получать информацию и от клавиш мыши, и с клавиатуры (обычно используемой только для ввода текста, но иногда и для сигнальных функций). Ввод с клавиатуры происходит через процедуру BusyRead (ЧтениеЗанятого), которая в отличие от стандартной процедуры Read не ждет очередного нажатия клавиши, а немедленно возвращает значение 0C, если символ отсутствует.



```
MODULE Рисование;
 FROM Terminal IMPORT BusyRead;
 FROM LineDrawing IMPORT
  width,height,Px,Py,dot,line,area,clear;
 FROM Mouse IMPORT
  keys,Mx,My,FlipCursor,TrackMouse,ShowMenu;

 CONST L = 512; (* размер квадрата *)
  ESC = 33C; DEL = 177C;

 VAR i,цвет,x0,y0,x1,y1: INTEGER;
  minx,maxx,miny,maxy: INTEGER;
  ch: CHAR;

 PROCEDURE ИнициализацияЭкрана;
 BEGIN area(1,0,0,width,height);
  Px := minx; Py := miny; area(0,Px,Py,L,L);
  line(0,L); line(2,L); line(4,L); line(6,L);
 END ИнициализацияЭкрана;

BEGIN
 minx := (width-L) DIV 2; miny := (height-L) DIV 2;
 maxx := minx + L; maxy := miny + L; цвет := 3;
 ИнициализацияЭкрана; FlipCursor;(* включить курсор *)
 REPEAT TrackMouse;
  IF 14 IN keys THEN
   ShowMenu("white|grey0|grey1|black",i);
   IF i # 0 THEN цвет := i - 1 END
  ELSIF (15 IN keys)&(minx <= Mx)&(Mx < maxx)
      &(miny <= My)&(My < maxy) THEN
   x1 := (Mx - minx) DIV 8; y1 := (My - miny) DIV 8;
   IF (x1 # x0) OR (y1 # y0) THEN
    FlipCursor; (*выключить*)
    area(цвет,minx + x1*8,miny + y1*8,8,8);
    x0 := x1; y0 := y1;
    FlipCursor (*включить*)
   END
  END;
  BusyRead(ch);
  IF ch = DEL THEN
   FlipCursor; ИнициализацияЭкрана; FlipCursor;
  END
 UNTIL ch = ESC;
 clear
END Рисование.
```

Обратимся теперь к модулю, позволяющему имитировать наличие многих дисплеев на единственном экране. Каждый имитируемый дисплей представляется прямоугольником, в котором операции

вывода графики и текста так же доступны, как и для всего экрана. Такой прямоугольный участок экрана называется окном, поскольку он может считаться окном, через которое может быть видна выбранная часть документа. Эта техника была использована в программе преобразования выражений в польскую инверсную запись. Эта программа описана в разделе, посвященном рекурсии (разд.14).

При использовании модуля работы с окнами можно открывать (создавать) окна и закрывать их по мере необходимости. Каждое окно изображается в виде прямоугольной области, содержащей заголовок, задаваемый при открытии окна. Окна можно перемещать, как если бы это были листы бумаги, лежащие на столе, размеры окон можно менять, и их можно накладывать одно на другое, опять же как будто бы это листы бумаги, лежащие на столе. Следовательно, модуль работы с окнами дает мощное средство одновременного просмотра и обработки многих документов и тем самым очень существенно увеличивает возможности экрана. Улучшение становится еще значительнее, если при работе с окнами возможен выбор размеров и начертания литер, причем буквы большего размера употребляются в наиболее используемых окнах, а меньшего – в менее важных документах или при необходимости просмотра за один раз больших фрагментов текста.

Хороший пример одновременного использования нескольких окон – отладчик программ. На приведенном в разд. 29 рисунке окна используются для просмотра текста программы, значений переменных, содержимого памяти, последовательности вызовов, а также для диалога между программистом и ЭВМ.

Так как экран предоставляет гораздо более широкие возможности, чем просто последовательный вывод текста, то будет разумным разбить систему работы с окнами на несколько модулей. Они позволят выбрать только нужные средства (и исключить те, которые не потребуются). Основной модуль может, например, сосредоточить возможности, необходимые во всех приложениях, такие, как создание, удаление и наложение окон. Такой модуль должен обеспечивать вычерчивание рамок окон в соответствующих местах. Дополнительные модули могут затем по отдельности обеспечивать либо возможность записи текста, либо возможность вычерчивания прямых линий и заполнения областей шаблонами. Пример такой схемы приведен в приложении. Там даны два модуля Windows и TextWindows. Основной модуль предоставляет процедуры для открытия нового окна и для закрытия окна. Кроме того, может изменяться как размер, так и положение окна процедурой RedefineWindow. В третьем измерении окно можно помещать на верх, или в самый низ перекрывающихся окон. Этот метод позволяет сделать доступным больше окон (портов данных), чем действительно могло бы поместиться на экране.

Модуль TextWindows обеспечивает средства для записи последовательного текста, во многом аналогичные средствам основного модуля **Terminal**. Кроме того, он позволяет задавать позицию записи (**SetPos**), получать эту позицию (**GetPos**),

устанавливать текстовый курсор в заданную позицию (SetCaret) и инвертировать область, в которую идет запись. Еще можно задавать действие, выполняемое при достижении конца страницы (AssignEOWAction). Это действие передается в процедуру как параметр процедурного типа. Такой способ может служить прекрасной иллюстрацией полезности понятия формальной процедуры. Ее роль еще больше в случае перемещений окна и изменения его размеров. В этом случае в модуле вызывается процедура-параметр, полученная ранее при вызове AssignRestoreProc. Такая техника работы позволяет модулю не обращать внимание на реальное содержимое его окон: ответственность за их восстановление ложится на пользователя. В модуле просто определяется момент, когда нужно восстановить окно.

Как уже показано в простом модуле Mouse, наличие дисплея с высоким разрешением экрана делает очень привлекательным метод ввода данных с помощью указательного устройства, позиция которого задается курсором. Наличие курсора требует значительной интеграции модуля работы с окнами с модулем управления мышью. Модуль CursorMouse, реализующий эту связь, тоже приведен в приложении.

И наконец, завершается пакет модулем Menu, который тоже тесно связан со вводом мышью и предлагает метод иерархического меню.

В заключение мы попытаемся сформулировать несколько полезных правил, которых нужно придерживаться при разработке программ, ведущих диалог с пользователем. Диалог осуществляется с использованием для ввода клавиатуры и указательного устройства (мыши), а для вывода применяется экран либо в оконном режиме, либо нет.

1. Всякому вводу текста должен предшествовать вывод строки, указывающей смысл ответа.

2. Следует учитывать исправления, вносимые с помощью клавиши забоя DEL, а также требование завершения ввода, задаваемое с помощью определенных клавиш, таких, как RETURN (возврат каретки) или пробел.

3. Определенная клавиша (обычно ESC – выход) должна быть зарезервирована для завершения программы.

4. Если используется указательное устройство, клавиатура резервируется для ввода текста (который обычно служит параметром команды), а сами команды вводятся, как правило, с использованием меню.

5. Меню должны быть короткими, не превышать 8 команд. Помните, что выдаваемое меню может зависеть от текущей позиции курсора, т.е. непосредственно относиться к указываемому объекту или окружению.

Часть 5

29. СРЕДСТВА ПРОГРАММИРОВАНИЯ НИЗКОГО УРОВНЯ

Языки высокого уровня поощряют и даже принуждают программиста разрабатывать программы структурированными. Структурные операторы обеспечивают высокую степень четкости и ясности текста программируемого алгоритма. Структурированные описания допускают высокий уровень абстракции при организации данных в программе и установлении соответствия данных с понятиями конкретной проблемной области. Главное преимущество таких языков – дополнительные средства контроля ошибок, поскольку структурность обеспечивает избыточность, которая может (и должна) использоваться реализаторами (в особенности компиляторов) для определения несоответствий в программе, которые проявляются как нарушение правил языка. В связи с этим понятие типов данных оказывается особенно мощным средством, а значит, и важной особенностью языков высокого уровня.

Мы, однако, осознаем, что существуют приложения, в которых правила языка в том виде, как это изложено в предыдущих разделах, оказываются слишком обременительными. Обычно это те приложения, где данные некоторой определенной структуры должны рассматриваться как имеющие другую структуру, т.е. там, где представление данных не определено заранее с помощью описания, заданного языком высокого уровня. Сюда включаются также те случаи, когда структура данных должна удовлетворять условиям, накладываемым спецификой конкретной ЭВМ, короче, когда должны учитываться машинные зависимости. Структура данных, выдаваемых программой, написанной на другом языке, либо программой на Модуле, но работающей на ЭВМ другого типа, обычно описывается в терминах машинно-зависимых объектов.

Еще один случай возникает, когда программы должны быть написаны для машин, в которых некоторые адреса памяти зарезервированы для особых целей. Если необходим доступ к ячейкам с этими адресами, то мы должны иметь возможность указать их расположение в памяти.

Модула-2 как универсальный язык программирования предназначен также и для решения задач вышеупомянутого типа и, следовательно, должен обеспечивать соответствующие средства. Они называются средствами программирования низкого уровня, поскольку позволяют

проводить рассмотрение на низком уровне абстракции, близком к используемой машине. Следовательно, их дальнейшее рассмотрение по самой их природе должно быть неполным. Мы можем просто задать их общую форму, а детали должны содержаться в документации, описывающей конкретную реализацию.

Главная особенность средств низкого уровня в том, что в них отсутствует избыточность, а следовательно, не происходит проверок соответствия правилам языка. Таким образом, программист, использующий такие средства, гораздо меньше защищен от ошибок. Поэтому настоятельно рекомендуется применять эти средства только там, где это действительно неизбежно, поскольку многие проблемы могут быть решены и привычными средствами языка.

Основным из этих средств является способ обхода контроля типов Модулы: идентификатор типа может быть использован в качестве идентификатора функции и обозначать операцию приведения. Считается, что *такая функция преобразует значение* типа, задаваемого параметром, в соответствующее значение того типа, который задан идентификатором функции. Например, значение выражения c типа CARDINAL отображается в соответствующее значение типа INTEGER функцией INTEGER(c). Функция BITSET(c) обеспечивает соответствующее значение типа BITSET. Эти соответствия не определяются языком Модула: информация зависит от машины и является поэтому дополнительной. Ключевая идея таких преобразований типа состоит в том, что они не содержат никаких реальных вычислений. Если нужно битовое представление значения c проинтерпретировать как целое, то используется INTEGER(c), а если его нужно интерпретировать как множество битов в слове, то BITSET(c). Программист, следовательно, должен знать особенности внутреннего представления рассматриваемого типа данных на машине. Описываемые функции предназначены для указания желаемого способа интерпретации значения и для отключения контроля типов.

Среди средств программирования низкого уровня имеются также еще два типа данных, которые следует обсудить детальнее. Эти типы называются WORD (слово) и ADDRESS (адрес). Память любой ЭВМ — это последовательность *так называемых слов, причем каждое* слово является отдельно адресуемой единицей, состоящей из фиксированного числа битов. Данные, описываемые в Модуле, отображаются компилятором в одно или несколько слов. Модула не разрешает применять к типу WORD никаких операций (кроме *присваивания*), поскольку такое значение считается неинтерпретируемым. Однако использование упоминаемых выше функций преобразования типов дает возможность применять операции и к словам, так как эти функции указывают желаемую интерпретацию. Очевидно, что такое использование типа WORD приводит к большой зависимости программы от реализации, поскольку на некоторых машинах слово может состоять всего лишь из 8 битов. Использование этого типа автоматически делает программу немобильной.

Если формальный параметр процедуры имеет тип WORD, то

соответствующий ему фактический параметр может иметь любой тип, занимающий одно слово памяти. В таком случае в вызове не требуется указывать явно преобразование типа. Например, процедуры ReadWord и WriteWord модулей Streams и FileSystem, введенных ранее, определяют параметр типа WORD. Они позволяют читать и писать последовательность слов и интерпретировать слова в соответствии с типом подставляемых параметров. В разных вызовах параметры могут иметь разный тип. Имея это средство, можно читать файлы, содержащие данные, форматированные по правилам, не выражаемым в терминах структур данных Модулы. В качестве простого примера рассмотрим задачу чтения файла, первое слово которого содержит его длину, а за этим словом следуют пары слов, первый элемент которых — это число, а второй имеет тип BITSET. Считая, что как тип CARDINAL, так и тип BITSET занимают одно слово, эту задачу можно записать так:

```
ReadWord(in,длина);
длина := длина - 1;
WHILE длина > 1 DO
 ReadWord(in,число); ReadWord(in,множество);
 обработать(число,множество); длина := длина - 2
END
```

Правило совместимости формального параметра типа WORD с любым типом фактического параметра (если тот занимает в памяти одно слово) обобщается в случае гибкого массива типа WORD. А именно если формальный параметр задан как ARRAY OF WORD, то любая переменная, структурированная или неструктурированная, может быть подставлена в качестве фактического параметра. Использование этого средства требует точного знания реализуемого компилятором способа отображения структуры данных в последовательность слов. Размер, т.е. число слов, занимаемых переменной v, определяетея функцией SIZE(v), а размер любой переменной типа T — функцией TSIZE(T). SIZE является стандартной функцией Модулы, а TSIZE должна импортироваться из стандартного модуля SYSTEM.

Тип ADDRESS обозначает величины, используемые в качестве адресов слов, и он определен как

```
TYPE ADDRESS = POINTER TO WORD
```

Слово, на которое ссылается адрес a можно теперь обозначить a^, где ^ — та же операция разыменования, используемая для обычных указателей. Значения типа ADDRESS считаются совместимыми по присваиванию с указателями любого типа. Это правило особенно важно в случае параметров: если формальный параметр имеет тип ADDRESS, то соответствующий ему фактический параметр может иметь любой указательный тип. Вследствие этого, как и в случае типа WORD, не происходит проверок соответствия типов. К операндам

```
program
 1 000044 IMPLEMENTATION MODULE SilDisplay;  (*
 2 000044   FROM SYSTEM IMPORT WORD, ADDRESS
 3 000044   FROM Program IMPORT AllocateHeap;
 4 000044   FROM BitmapVars IMPORT BMD;
 5 000044   FROM FileSystem IMPORT File, Lookup, Re
 6 000044
 7 000044   CONST Height = 800;
 8 000044
 9 000044   TYPE
10 000044     DispMode = (replace, paint, invert, erase
11 000044     BlockDescriptor =
12 000044       RECORD x,y,w,h: INTEGER
13 000044       END ;
14 000044     Pattern =
15 000044       RECORD length: CARDINAL;
16 000044              w: ARRAY [0..15] OF BITSET;
17 000044       END ;
```

```
access
045615
nge error
045642  PC
 033073  error
045653  mask
 167466  Tmask
```

```
proced
 7 Co
 8 Mo
 9 Ca
10 Fi
11 Te
12 De
13 SI
14 Si
```

```
memory
036010                                                                041742
036020  042042  042342  000000  000000  000000  000000  000000  000000
036030  000000  000000  000000  000000  000000  000000  000000  000000
036040  000000  000000  000000  000000  000000  000000  000000  000000
```

```
AY              at 033117
ursor       4 RECORD    at 033133
llmap       4 RECORD    at 033137
lk          4 RECORD    at 033143
ursorPat0  17 RECORD    at 033147
ursorPat1  17 RECORD    at 033170
ickPat     17 RECORD    at 033211
at         68 ARRAY     at 033232
aveBlk      4 RECORD    at 033336
aveBMD  65300 CARDINAL  at 033115
```

```
dialog
which window
new font>
TIMESROMAN16.FONT  done
change which window
change window   program
point the diagonal
which window
 write picture >Fig1.PICT
```

типа ADDRESS, кроме того, могут применяться арифметические операции. В зависимости от конкретной реализации Модулы, тип ADDRESS совместим с некоторым арифметическим типом, например CARDINAL, INTEGER, LONGINT или LONGCARD. Это средство, помимо всего прочего, позволяет записывать программы управления памятью. Предположим, например, что последовательность слов должна читаться из файла и загружаться в память с адресами нач, нач+1,... . Пусть опять первое слово задает длину последовательности. Функция ADR(x) дает адрес переменной x, где x может иметь любой тип.

```
ReadWord(in,длина);
длина := длина - 1; a := ADR(буфер);
WHILE длина > 0 DO
 ReadWord(in,a^); a := a + 1; длина := длина - 1
END
```

Типы WORD и ADDRESS тоже импортируются из модуля SYSTEM. Это значит, что в заголовках программ, использующих эти типы низкого уровня, будет явное упоминание об этом. Модуль SYSTEM содержит типы данных и связанные с ними процедуры, обработка которых происходит по особым правилам, известным компилятору. Следовательно, этот модуль тесно связан с конкретным

компилятором и его нельзя описать в виде отдельного модуля. По этой причине он называется псевдомодулем. Но можно тем не менее считать, что он описывается следующим модулем определений:

```
DEFINITION MODULE SYSTEM;
 TYPE WORD,ADDRESS;

 PROCEDURE ADR(x: ЛюбойТип): ADDRESS;
  (* адрес переменной x *)
 PROCEDURE TSIZE(ЛюбойТип): CARDINAL;
  (* длина объекта данного типа в словах *)

 PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS;
       n: CARDINAL; VAR q: ADDRESS);
 PROCEDURE TRANSFER(VAR кто,кому: ADDRESS);
  ...
END SYSTEM.
```

Точки означают, что в модуле могут содержаться и другие средства, зависящие от конкретной реализации.

Реализации Модулы могут, но не обязаны предоставлять возможность задать для переменной фиксированный адрес. Если эта возможность имеется, то адрес задается непосредственно за идентификатором переменной в ее описании. Примеры приводятся в следующем разделе.

30. ПАРАЛЛЕЛЬНЫЕ ПРОЦЕССЫ И СОПРОГРАММЫ

В этом разделе введем ряд понятий мультипрограммирования, т.е. программирования различных, параллельно выполняемых вычислений. Применение этих средств намеренно ограничено областью так называемых слабо связанных процессов. Мы исключаем из рассмотрения сильно связанные массивы процессов, считая эту область слишком узкой и специальной. Вместо этого ограничимся рассмотрением программ, описывающих несколько процессов, взаимодействующих сравнительно редко и поэтому называемых слабо связанными. Под словом "редко" подразумевается, что взаимодействия происходят в немногих четко определенных и явно выделенных точках программы, а слово "процесс" означает последовательность действий, т.е. принципиально последовательный процесс. Программирование в том виде, в котором мы его изучали до сих пор, может, следовательно, рассматриваться как частный случай программирования, включающий лишь один процесс. И наоборот, при мультипрограммировании можно использовать все

средства и методы, изученные до сих пор, и требуется лишь просто добавить несколько новых средств описания параллельных процессов и их взаимодействия. В этом отношении мы следуем традиции ранних языков мультипрограммирования, таких, как Модула-1 и Concurrent Pascal (Параллельный Паскаль), разработанный Бринчем Хансеном.

В общем случае следует выделять следующие типы систем мультипрограммирования.

1. ЭВМ содержит несколько одинаковых процессоров. Запрограммированные процессы выполняются в истинном параллельном режиме.

2. ЭВМ содержит единственный процессор. В каждый момент времени выполняется лишь один процесс, и время от времени происходит переключение процессов, т.е. мультиплексирование по времени. Более общий случай – когда в системе, содержащей m процессоров, выполняются n процессов, причем m обычно меньше, чем n. Такой режим называется квазипараллельным.

3. На нескольких процессорах с различными возможностями выполняются разные процессы. Некоторые из этих процессов таковы, что в них есть фрагменты, которые можно выполнять только на определенных процессорах. Типичным примером таких специализированных процессоров являются устройства ввода-вывода.

Наша цель – определить систему понятий и нотацию, которые позволят выразить общие аспекты всех трех перечисленных типов систем в одинаковых терминах и на высоком уровне абстракции. Можно с некоторой степенью приближения считать, что различие между типами 1 и 2 – это лишь вопрос реализации. Точнее, если мы запишем логические процессы и их взаимодействие таким образом, что они смогут выполняться в режиме истинной параллельности, то система, построенная на одном процессоре, может использоваться для выполнения этих процессов в квазипараллельном режиме. Случай 3 требует особого отношения, поскольку очевидно, что наличие процессоров со специфическими возможностями не может быть скрыто просто как деталь реализации.

В этой главе мы обсудим описание процессов и их взаимодействия в терминах Модулы-2. Кроме того, дадим реализацию, опирающуюся на однопроцессорную машину, и понятие сопрограммы, т.е. такую систему, которая реализует квазипараллельность. Программирование для специализированных устройств (ввода и вывода) и их работа в режиме истинной параллельности описаны в следующем разделе. Для описания параллельных процессов введен модуль Processes (процессы). Его достоинство в том, что он содержит средства мультипрограммирования на высоком уровне абстракции, причем для этого фактически не требуется никаких дополнительных языковых средств. Он предоставляет все возможности Модулы-1, а также большинство возможностей языка **Concurrent Pascal**.


```
DEFINITION MODULE Processes;
 TYPE SIGNAL;

 PROCEDURE StartProcess(P: PROC; n: CARDINAL);
  (* начать параллельный процесс, задаваемый
     программой P с рабочей областью размером n.
     PROC – стандартный тип, определенный как
     PROC = PROCEDURE *)
 PROCEDURE SEND(VAR s: SIGNAL);
  (* возобновляется один из процессов, ждущих s *)
 PROCEDURE WAIT(VAR s: SIGNAL);
  (* ждать,пока другой процесс не пошлет сигнал s *)
 PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
  (* Awaited(s) =
          "по крайней мере один процесс ждет s" *)
 PROCEDURE Init(VAR s: SIGNAL);
  (* обязательная инициализация *)
END Processes.
```

Вызов StartProcess(P,n) начинает выполнение процесса, который выражается процедурой P. Будет ли этот процесс выполняться параллельно или квазипараллельно, зависит, конечно, от реализации используемого модуля Processes. Каждый процесс требует рабочей области определенного размера для размещения своих локальных переменных. Размер рабочей области в словах задается параметром n. Его величина зависит от числа локальных переменных и локальных вызовов, использованных в этом процессе. (Размер типичной минимальной рабочей области – 100 слов.)

Связь между процессами осуществляется двумя различными способами: посредством общих, разделяемых переменных и посредством так называемых сигналов. Общие переменные используются для передачи данных между процессами, и здесь возникает проблема согласованного взаимодействия. Когда некоторый процесс осуществляет какие-либо действия над некоторой общей переменной, то нельзя, чтобы в этот момент ее использовал или менял еще какой-то процесс. Разумным решением проблемы было бы заключить общие переменные в модуль, который гарантировал взаимное исключение процессов. Такой модуль называется монитором, он будет обсуждаться ниже. Сигналы, экспортируемые как тип данных из модуля Processes, сами по себе не несут информации, а служат для синхронизации. К сигналам применимы две операции (не считая обязательной начальной инициализации): процесс может послать сигнал и может ждать сигнала (посланного другим процессом.) Каждый сигнал обозначает возникновение некоторого условия. Мы настоятельно рекомендуем указывать это условие в виде комментария при описании сигнала. Посылка сигнала возобновляет не более одного процесса. (В противном случае один из возобновляемых процессов мог бы быстро нарушить условие и тогда другие процессы работали бы с неверной предпосылкой,

считая условие истинным.) Посылка сигнала, если его не ждет ни один процесс, эквивалентна пустому оператору.

Программист должен понимать, не заботясь о деталях реализации, что в системах, использующих квазипараллельность, вызовы SEND и WAIT подразумевают (или могут подразумевать) переключение вызывающего процесса на другой (ждущий) процесс, и что это единственные места, где такое переключение может происходить. Следовательно, ожидание наступления определенного события нельзя программировать с использованием пустых циклов (так называемого ожидания занятого), а только используя явные вызовы WAIT.

Другое важное правило программирования состоит в том, что разделяемые переменные описываются и скрываются в мониторе. Монитор – модуль, который гарантирует взаимное исключение процессов и тем самым обеспечивает целостность своих локальных данных. Доступ к этим данным (поскольку они скрыты) ограничен лишь операторами вызова процедур (экспортируемых из) монитора, а так как монитор гарантирует, что процесс, вызывающий его процедуру, временно задерживается, пока другой процесс выполняет любую из процедур монитора, тем самым автоматически достигается взаимное исключение. Модуль специфицируется как монитор указанием в его заголовке приоритета. Значение приоритета – число типа CARDINAL. Здесь достаточно знать, что указание любого приоритета делает модуль монитором.

Следующий пример призван проиллюстрировать приведенные правила. Он решает одну из классических задач мультипрограммирования: задачу обмена данными между разными процессами. Обычно при решении таких задач используется буфер. Чем больше буфер, тем слабее процессы связаны. Мы считаем, что процессы могут помещать элементы данных в буфер и извлекать их оттуда. Буфер – принципиально разделяемая переменная. Вместе с процедурами Поместить и Извлечь он изолирован в мониторе. Поскольку мы не знаем (не должны и не хотим знать) частностей рассматриваемых процессов, обратимся непосредственно к ключевой проблеме мультипрограммирования, к монитору, через который осуществляется взаимодействие процессов. Сами процессы содержат вызовы процедур Поместить и Извлечь и обычно являются циклическими. Этими вызовами и ограничивается их взаимодействие. Если процесс содержит вызовы процедуры Поместить, то это поставщик. Процессы, содержащие вызовы Извлечь, называются потребителями. В нашем примере буфер описан как переменная-массив, используемая циклическим образом. Могут возникать два условия для ожидания: при вызове поставщиком процедуры Поместить буфер может оказаться полон, а при вызове потребителем Извлечь – пустым. Эти два условия приводят к описанию двух сигналов, называемых непуст и неполон, используемых для реактивации ждущих процессов. Монитор Буфер запрограммирован как локальный модуль. Переменная n содержит число элементов, находящихся в буфере.


```
MODULE Буфер [1];
 EXPORT Поместить,Извлечь;
 IMPORT SIGNAL,SEND,WAIT,Init,ТипЭлемента;

 CONST N = 128; (* размер буфера *)
 VAR n: [0..N]; (* число помещенных элементов *)
  неполон: SIGNAL; (* n < N *)
  непуст:  SIGNAL; (* n > 0 *)
  in,out: [0..N-1]; (*индексы*)
  буф: ARRAY [0..N-1] OF ТипЭлемента;

 PROCEDURE Поместить(x: ТипЭлемента);
 BEGIN
  IF n = N THEN WAIT(неполон) END;
  (* n < N *) n := n + 1; (* 0 < n <= N *)
  буф[in] := x; in := (in + 1) MOD N;
  SEND(непуст)
 END Поместить;

 PROCEDURE Извлечь(VAR x: ТипЭлемента);
 BEGIN
  IF n = 0 THEN WAIT(непуст) END;
  (* n > 0 *) n := n - 1; (* 0 <= n < N *)
  x := буф[out]; out := (out + 1) MOD N;
  SEND(неполон)
 END Извлечь;

BEGIN n := 0; in := 0; out := 0;
 Init(неполон); Init(непуст)
END Буфер
```

К сожалению, приведенная версия алгоритма буферизации имеет обыкновение посылать сигнал всякий раз, как только элемент помещается в буфер или извлекается из него. Однако, в принципе, синхронизирующие сигналы нужно посылать, только если есть процесс, который его действительно ждет. Хороший принцип – минимизировать число обменов сигналами и тем самым уменьшить степень связи процессов. Усовершенствование в этом направлении достигается в версии алгоритма, предложенной Дейкстрой и носящей название "алгоритм спящего парикмахера". В этой версии диапазон переменной n расширяется. Теперь в случае, когда в монитор не вошел ни один процесс, переменная n имеет такой смысл:

n < 0:	буфер пуст и −n потребителей в состоянии ожидания
0 <= n <= N:	буфер содержит n элементов и ожидающих процессов нет
N < n:	буфер полон и n − N поставщиков в состоянии ожидания

Процедуры Поместить и Извлечь описываются так:

```
PROCEDURE Поместить(x: ТипЭлемента);
BEGIN0 n := n + 1;
 IF n > N THEN WAIT(неполон) END;
 (* n <= N *)
 буф[in] := x; in := (in + 1) MOD N;
 IF n <= 0 THEN SEND(непуст) END
END Поместить;

PROCEDURE Извлечь(VAR x: ТипЭлемента);
BEGIN n := n - 1;
 IF n < 0 THEN WAIT(непуст) END;
 (* n >= 0 *)
 x := буф[out]; out := (out + 1) MOD N;
 IF n >= N THEN SEND(неполон) END
END Извлечь;
```

Теперь обсудим проблему представления модуля **Processes** и дадим одно из возможных решений. Подчеркнем, что это лишь одна из возможностей. Решение рассчитано на однопроцессорную машину, и этот единственный процессор разделяется во времени для обслуживания различных процессов. Решение основано на понятии сопрограммы. Сопрограмма – последовательная программа, по существу сходная с процессом, как он обсуждался выше. Принципиальные различия между процессом и сопрограммой следующие:

1. Известно, что сопрограммы выполняются квазипараллельно. Следовательно, их использование исключает трудную проблему взаимодействия истинно параллельных процессов.
2. Переключение процессора от одной сопрограммы к другой осуществляется явным оператором передачи управления. Выполнение сопрограммы, которой передается управление, возобновляется с той точки, где она была приостановлена последним таким оператором.

Очевидно, что на однопроцессорной машине процесс может быть реализован сопрограммой, возникающей на более низком концептуальном уровне. Ее близость к реальным ЭВМ становится очевидной, если сравнить оператор передачи управления Модулы с машинным оператором перехода. Такой сопрограммный переход должен запоминать текущее состояние выполняемой процедуры, о которой говорят, что она приостанавливается. Эта процедура может быть затем возобновлена, если другая сопрограмма передаст управление назад приостановленной. При выполнении операторов передачи управления сопрограмма указывается явно, в отличие от операторов **SEND** и **WAIT**, используемых для синхронизации процессов.

Поскольку в Модуле сопрограммы считаются средствами низкого уровня, связанные с ними типы и операции должны импортироваться из модуля **SYSTEM** (см. раздел, посвященный средствам низкого уровня) или из другого модуля низкого уровня. В частности, там имеются тип **ADDRESS** и процедура **TRANSFER**.

Заголовок процедуры передачи управления выглядит так:

```
PROCEDURE TRANSFER(VAR источн,приемн: ADDRESS);
```

Ее вызов приводит к задержке вызывающей сопрограммы (чтобы позже быть возобновленной, начиная с оператора, следующего за вызовом) и возобновлению сопрограммы, принимающей управление, с точки задержки. Для создания сопрограммы должна вызываться процедура **NEWPROCESS** (новый процесс).

```
PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS;
            n: CARDINAL; VAR Нов: ADDRESS);
```

Идентификатор P обозначает процедуру без параметров, представляющую программу для вновь создаваемой сопрограммы; A – начальный адрес рабочей области, в которой размещаются локальные переменные сопрограммы и запоминается состояние сопрограммы при задержке; n – объем рабочей области в единицах памяти. Вызов присваивает переменной Нов ссылку на вновь создаваемую сопрограмму, причем ее состояние инициализируется так, что при передаче ей управления выполнение начинается от начала процедуры P. Следовательно, сопрограммы начинаются явной передачей управления и должны заканчиваться тоже такой передачей.

Теперь можно представить реализацию модуля **Processes** в терминах сопрограмм. Существенный аспект состоит в том, что вызовы **WAIT** и **SEND** должны транслироваться в операторы передачи, в которых необходимо указывать, куда именно передается управление. Следовательно, модуль **Processes** должен включать в себя административную систему, которая справедливо распределяет процессорное время между процессами. Такая административная система называется диспетчером. Обычно это часть операционной системы ЭВМ. В действительности конкретные реализации могут запрещать введение сопрограмм и предоставлять только средства высокого уровня, содержащиеся в модуле **Processes**.

```
IMPLEMENTATION MODULE Processes [1];
 FROM SYSTEM IMPORT ADDRESS,TSIZE,NEWPROCESS,TRANSFER;
 FROM Storage IMPORT Allocate;

 TYPE SYGNAL = POINTER TO ДескрипторПроцесса;
  ДескрипторПроцесса =
   RECORD следующий: SIGNAL; (* кольцевой список *)
    очередь: SIGNAL; (* очередь ждущих процессов *)
    сопрг: ADDRESS;
    готов: BOOLEAN
   END;
```

```
VAR ТекПроцесс: SIGNAL; (* текущий процесс *)

PROCEDURE StartProcess(P: PROC; n: CARDINAL);
 VAR s0: SIGNAL; P0бл: ADDRESS;
BEGIN s0 := ТекПроцесс; Allocate(P0бл,n);
 Allocate(ТекПроцесс,TSIZE(ДескрипторПроцесса));
 WITH ТекПроцесс^ DO
  следующий := s0^.следующий;
  s0^.следующий := ТекПроцесс;
  готов := TRUE; очередь := NIL
 END;
 NEWPROCESS(P,P0бл,n,ТекПроцесс^.сопрг);
 TRANSFER(s0^.сопрг,ТекПроцесс^.сопрг)
END StartProcess;

PROCEDURE SEND(VAR s: SIGNAL);
 VAR s0: SIGNAL;
BEGIN
 IF s # NIL THEN
  s0 := ТекПроцесс; ТекПроцесс := s;
  WITH ТекПроцесс^ DO
   s := очередь; готов := TRUE; очередь := NIL
  END;
  TRANSFER(s0^.сопрг,ТекПроцесс^.сопрг)
 END
END SEND;

PROCEDURE WAIT(VAR s: SIGNAL);
 VAR s0,s1: SIGNAL;
BEGIN (* вставить ТекПроцесс в очередь s *)
 IF s = NIL THEN s := ТекПроцесс
 ELSE s0 := s; s1 := s0^.очередь;
  WHILE s1 # NIL DO
   s0 := s1; s1 := s0^.очередь
  END;
  s0^.очередь := ТекПроцесс
 END;
 s0 := ТекПроцесс;
 REPEAT ТекПроцесс := ТекПроцесс^.следующий
 UNTIL ТекПроцесс^.готов;
 IF ТекПроцесс = s0 THEN (*тупик*) HALT END;
 s0^.готов := FALSE;
 TRANSFER(s0^.сопрг,ТекПроцесс^.сопрг)
END WAIT;

PROCEDURE Awaited(s: SIGNAL): BOOLEAN;
BEGIN RETURN s # NIL
END Awaited;
```



```
 PROCEDURE Init(VAR s: SIGNAL);
 BEGIN s := NIL
 END Init;

BEGIN Allocate(ТекПроцесс,TSIZE(ДескрипторПроцесса));
 WITH ТекПроцесс^ DO
  следующий := ТекПроцесс; готов := TRUE; очередь := NIL
 END
END Processes.
```

При запуске процесса вызовом StartProcess(P,n) происходит выделение памяти под дескриптор (описатель) процесса и рабочую область связанной с ним сопрограммы. Дескриптор вставляется в кольцевой список, содержащий дескрипторы всех процессов, созданных до сих пор. Переменная ТекПроцесс указывает на дескриптор текущего выполняемого процесса. Просмотром кольцевого списка можно добраться до любого процесса. Процесс-преемник обозначается компонентой следующий дескриптора.

Ключевым вопросом является способ представления сигналов. В то время как на уровне абстракции пользователя сигнал представляет возникающее условие, на уровне реализации – это множество процессов, ждущих сигнала. Так как число этих процессов неизвестно, будет разумным решением организовать их в связанный список. Следовательно, переменная типа SIGNAL представляет собой голову списка, а каждый дескриптор процесса содержит компоненту очередь, связывающую его со следующим процессом, ждущим сигнала. Если ждущих процессов нет, то его значение равно NIL.

Из приведенного описания становятся очевидными функции процедур SEND и WAIT. SEND берет первый элемент списка s и возобновляет соответствующий ему процесс. Процесс, посылающий сигнал (на его дескриптор указывает переменная ТекПроцесс), приостанавливается. WAIT ставит вызывающий ее процесс (на его дескриптор указывает ТекПроцесс) в конец списка ждущих процессов. Вставка производится в конец, чтобы удовлетворить требованию справедливости распределения процессорного времени и обеспечить невозможность одному процессу обогнать другой, ждущий тот же сигнал, поскольку список реализует дисциплину очереди. Получается так, что, в принципе, ни один процесс, не ждущий сигнала, не может быть возобновлен. Но здесь справедливость достигается благодаря передвижению по кольцевому списку, начиная с ТекПроцесс. Дополнительный признак, называемый готов, используется для быстрого определения того, готов ли процесс для возобновления. (Это решение предпочтительнее того, когда ждущие процессы убираются из кольцевого списка, а затем требуют повторной вставки при реактивации. Предпочтение основывается на предположении, что число процессов не слишком велико.)

В принципе взаимодействие процессов должно быть заключено внутрь монитора, т.е. модуля, гарантирующего взаимное

исключение. Однако, поскольку мы условились, что этот модуль реализуется на однопроцессорной машине, параллельное взаимодействие невозможно по определению и, следовательно, описание монитора, т.е. указание приоритета в заголовке модуля, избыточно.

31. УПРАВЛЕНИЕ ВНЕШНИМИ УСТРОЙСТВАМИ, ПАРАЛЛЕЛЬНОСТЬ И ПРЕРЫВАНИЯ

В предыдущем разделе мы обсудили системы с несколькими процессами и то, как имитировать параллельные процессы, разделяя между ними во времени один процессор. Теперь мы рассмотрим обратную ситуацию, когда несколько процессоров участвуют в выполнении единственного процесса. Рассмотрим для простоты циклический процесс, состоящий из двух частей – поставщика и потребителя. Пусть этот процесс изображается так:

```
LOOP произвести(x); потребить(x) END
```

Теперь допустим, что каждая часть должна выполняться особым процессором. Понятно, что в каждый момент только один из двух процессоров может быть активным. Следовательно, их надо синхронизировать, что легко достигается введением синхронизирующей переменной s, имеющей смысл "потребитель активен" (с начальным значением FALSE). Каждый из этих двух процессоров описывается своей собственной программой, которые в свою очередь тоже циклические.

```
Поставщик:
 LOOP ждать(NOT s);
  произвести(x); s := TRUE
 END

Потребитель:
 LOOP ждать(s);
  потребить(x); s := FALSE
 END
```

Операцию wait(b) можно понимать как эквивалентную оператору

```
REPEAT (*опрос*) UNTIL b
```

Переменные x и s образуют интерфейс между процессорами. Эти переменные обычно реализуются как специальные аппаратные регистры, называемые <u>регистрами устройств</u>. В некоторых ЭВМ доступ к ним осуществляется специальными командами (в соответствующих реализациях Модулы доступ к этим регистрам осуществляется с помощью специальных процедур). В других ЭВМ эти

регистры реализованы как ячейки памяти с фиксированными адресами (так называемый <u>ввод-вывод через адреса памяти</u>).

В качестве примера рассмотрим взаимодействие процесса ввода с клавиатуры с "регулярным" процессом-потребителем, запрограммированным для ЭВМ PDP-11. Эта машина использует ввод-вывод через адреса памяти. Ее переменная состояния клавиатуры s, например, представляется седьмым битом слова памяти с адресом 777560B, а буферная переменная x – 0..7 битами слова с адресом 777562B. Поскольку в PDP-11 может (в принципе) быть много таких интерфейсных регистров, то соответствующая реализация предоставляет возможность задать в описании адрес переменной, как в приведенном ниже примере. Мы <u>настоятельно рекомендуем</u> программистам ограничить использование этого средства переменными, представляющими собой регистры устройств, и избегать злоупотребления ими для других целей. Два соответствующих регистра вводятся следующими описаниями:

```
VAR s[777560B]: BITSET;
 x[777562B]: CHAR;
```

Тогда как программа-поставщик реализована на аппаратном уровне, программа-потребитель записывается по следующей схеме:

```
LOOP
 REPEAT UNTIL 7 IN s;
 потребить(x)
END
```

Отсутствие оператора "s := TRUE" объясняется тем, что интерфейс клавиатуры спроектирован так, что обращение к x автоматически устанавливает s.

На этом завершим описание простого примера работы с клавиатурой, использующего циклический опрос устройства.

Недостаток описанной схемы в том, что процессы слишком тесно связаны, они жестко чередуются. Пока один активен – другой бездействует. Часто требуется, чтобы процессы не были так тесно связаны. Это достигается использованием буфера, причем желательно, чтобы его размер был побольше (в разумных пределах). Для работы буфера мы используем уже описанную в предыдущей главе схему поставщик – потребитель. И тот и другой представляются сопрограммами.

Здесь возникает следующий принципиальный вопрос: когда должны происходить передачи управления от сопрограммы к сопрограмме, т.е. обмен сигналами, так, чтобы обеспечить обоим процессорам (центральному и контроллеру клавиатуры) максимальную активность при минимальном возможном взаимодействии?

Для того чтобы отчетливее представить проблему, обратимся опять к конкретному примеру, включающему клавиатуру в качестве поставщика. Процесс-потребитель чередует действия по извлечению

элементов из буфера и их потреблению. Оба оператора выполняются центральным процессором общего назначения. Поставщик чередует выдачу элемента и помещение его в буфер. Последняя операция может выполняться центральным процессором. А вот выдача элемента осуществляется клавиатурой. Мы считаем (и это является характеристикой слабо связанных процессов), что занесение и извлечение элементов занимают пренебрежимо малое время по сравнению с выдачей элементов клавиатурой и их обработкой (потреблением). Следовательно, можно считать, что процесс-поставщик выполняется процессором клавиатуры, лишь иногда требуя обслуживания центральным процессором, который можно безболезненно отвлекать от его главной задачи ввиду пренебрежимо малого времени, которое он будет занят на этой вспомогательной работе. Теперь мы в состоянии ответить на заданный ранее вопрос.

Программируемый процессор переключается с процесса-потребителя всякий раз, когда клавиатура выполнила свою часть процесса-поставщика, и переключается обратно, как только операция помещения элемента в буфер заканчивается.

Обратное переключение на процесс-потребитель можно выполнить явно оператором TRANSFER. Однако его нельзя применить при переключении от потребителя к поставщику, поскольку место, в котором будет находиться процесс в момент переключения, заранее неизвестно. Фактически мы должны уметь прервать процесс-потребитель в любой точке. Или, выражаясь в терминах Модулы, должны уметь вставить оператор передачи управления в произвольное место программы, не задавая его заранее.

Большинство ЭВМ предлагает именно эту возможность — непрограммируемую передачу управления, называемую прерыванием. Для иллюстрации ее применения запрограммируем два процесса, использованные нами в качестве примера. Будем считать, что использование литер происходит в главной программе, мы же будем интересоваться лишь процедурой взять выборки литеры из буфера. Поставщик оформлен в виде сопрограммы, взаимодействующей с клавиатурой через интерфейсную переменную x. Совместно с частью процесса-поставщика, помещающей литеру в буфер, эта переменная описывает интерфейс между процессами и, следовательно, заключена внутри монитора, скрывая буфер (см. предыдущий раздел). В то время как операция выборки литеры из буфера оформлена в виде процедуры, действия по помещению в буфер вставлены непосредственно в текст сопрограммы-поставщика, которая целиком содержится внутри монитора, поскольку выработка новой литеры клавиатурой не считается ограниченной требованием взаимного исключения процессов. Эта сопрограмма представляет собой то, что часто называют обработчиком прерываний.

```
MODULE Клавиатура [4];
 IMPORT ADR,SIZE,WORD,NEWPROCESS,TRANSFER,IOTRANSFER;
 EXPORT взять,n(*только на чтение*);
```



```
 CONST N = 32;
 VAR x[777562B]: CHAR; (*данные клавиатуры*)
  s[777560B]: BITSET;  (*состояние клавиатуры*)

 VAR n,in,out: CARDINAL;
  буфер: ARRAY [0..N-1] OF CHAR;
  ПОСТ,ПОТР: ADDRESS;
  РОбл: ARRAY [0..177B] OF WORD;

 PROCEDURE взять(VAR ch: CHAR);
 BEGIN (* должна вызываться только при n > 0 *)
  IF n > 0 THEN
   ch := буфер[out]; out := out + 1 MOD N;
   n := n - 1
  ELSE ch := 0C
  END
 END взять;

 PROCEDURE поставщик; (* работает как сопрограмма *)
 BEGIN
  LOOP IOTRANSFER(ПОСТ,ПОТР,60B);
   (* нажатие клавиши на клавиатуре вызывает действие,
      подобное операции TRANSFER(ПОТР,ПОСТ) в текущей
      точке программы-потребителя (= прерывание), и
      поставщик возобновляется в этой точке *)
   IF n < N THEN
    буфер[in] := x; in := (in + 1) MOD N;
    n := n + 1
    (* игнорировать литеры, если буфер полон *)
  END END
 END поставщик;

BEGIN n := 0; in := 0; out := 0;
 NEWPROCESS(поставщик,ADR(РОбл),SIZE(РОбл),ПОСТ);
 EXCL(s,6); TRANSFER(ПОТР,ПОСТ)
END Клавиатура
```

Приведенный модуль описывает обработку литер с клавиатуры для ЭВМ PDP-11. Он использует средства, зависящие от реализации, в частности интерфейсные переменные x и s. Необходимо упомянуть три детали:

1. Передача управления из прерывающей сопрограммы (поставщик) в прерываемую (потребитель) должна осуществляться оператором

IOTRANSFER(источник,приемник,ВектПрер)

где ВектПрер — дополнительный параметр, обозначающий зависящий

от аппаратуры адрес так называемого вектора прерывания.

2. Для каждого устройства должна быть явным образом включена возможность прерывания. Это осуществляется оператором EXCL(s,6), который сбрасывает бит запрета прерывания в регистре состояния клавиатуры. Начиная с этого момента, прерывание будет возникать всякий раз, когда будет нажиматься какая-либо клавиша на клавиатуре.

3. Важная возможность – запрещение непрограммируемого прерывания в тех точках, где оно могло бы оказаться опасным. Прерывания должны запрещаться во время выполнения критических операций над разделяемыми переменными. Следовательно, все такие операции должны быть помещены внутрь монитора, который бы гарантировал непрерываемость его частей. Это достигается указанием приоритета прерывания, который жестко определяется обслуживаемым внешним устройством; для клавиатуры PDP-11 приоритет равен четырем. (Отметим, что простой прием достижения взаимного исключения – запрещение прерываний.)

4. ЭВМ, имеющие так называемую систему приоритетных прерываний, позволяют выключать сигналы прерываний избирательно, в зависимости от назначенных приоритетов. Каждый источник прерываний имеет свой фиксированный приоритет q. Процессор же имеет не просто состояния запрета и разрешения прерываний, а уровень прерывания p, подразумевающий, что процессор может быть прерван только сигналами с приоритетом q > p.

На этом примере ввода с клавиатуры закончим демонстрацию использования сопрограмм и непрограммируемых передач управления. Добавим еще, что описанный здесь циклический процесс часто называют обработчиком прерываний, циклическая природа которого скрыта содержащимся в нем оператором передачи управления от сопрограммы к сопрограмме и последующим возвратом управления. Мы отдаем радикальное предпочтение явной записи циклического процесса и в особенности подчеркиваем, что удобнее всего рассматривать прерывания как непрограммируемые передачи управления.

Мы должны также упомянуть, что во многих вычислительных системах работа с внешними устройствами (а следовательно, и использование прерываний) является прерогативой резидентной операционной системы. В таких ЭВМ программист не должен пользоваться этими средствами, даже если такие нарушения и могут остаться незамеченными, поскольку эти действия могут представлять серьезную угрозу правильной работе операционной системы, а значит, и другим ее пользователям. Но, поскольку Модула была задумана и как инструмент создания операционных систем, включение соответствующих средств работы с устройствами и прерываниями было неизбежно. Их использование, однако, должно быть ограничено так называемыми автономными системами, которые не поддерживаются (и не обременены) конкретной операционной системой.

Сообщение о языке программирования Модула-2

1. ВВЕДЕНИЕ

Язык Модула-2 возник из практических нужд как универсальный и эффективный язык системного программирования для мини-ЭВМ. Его предшественниками были Паскаль и Модула. От последнего он унаследовал имя, важное понятие модуля и систематический современный синтаксис, а от первого – почти все остальное. Сюда входят, в частности, структуры данных, т.е. массивы, записи, записи с вариантами, множества и указатели. В число структурных операторов входят условный оператор, оператор выбора, цикл с условием окончания, цикл с условием продолжения и оператор присоединения. Синтаксис структурных операторов таков, что каждый оканчивается специальным символом.

Язык по существу машинно-независим. Исключение составляют ограничения, связанные с длиной слова. Может показаться, что это противоречит понятию языка системного программирования, в котором должна быть возможность выразить все операции базовой машины. Эта дилемма разрешается при помощи понятия модуля. Машинно-зависимые понятия вводятся при помощи специальных модулей, поэтому их использование может быть очерчено определенными рамками. В частности, в этих случаях язык дает возможность ослабить правила совместимости типов. На хорошем языке системного программирования должна быть возможность написать процедуры преобразования при вводе-выводе, подпрограммы обработки файлов, распределения памяти, управления процессами и другие. Поэтому такие возможности должны быть не элементами самого языка, а модулями (как говорят, низкого уровня), являющимися компонентами большинства написанных программ. Поэтому такой набор стандартных модулей является существенной частью реализации Модулы-2.

Вместо процессов и их синхронизации с помощью сигналов, введенных в Модуле, в Модуле-2 используется низкоуровневое понятие сопрограммы. При этом, однако, можно сформулировать (стандартный) модуль, реализующий эти процессы и сигналы. Преимущество того, что они не включаются собственно в язык, состоит в том, что программист, запрограммировав самостоятельно соответствующий модуль, имеет возможность выбрать такой алгоритм управления процессами, который отвечает его нуждам. В простых

(но часто встречающихся) случаях, например когда взаимодействующие процессы выступают только как драйверы устройств, эти программы могут быть вообще опущены.

Современный язык системного программирования должен поддерживать разработку больших программ, возможно, создаваемых несколькими людьми. Модули, написанные каждым из них, имеют хорошо специфицированные интерфейсы, формулируемые независимо от реализации. В Модуле-2 это поддерживается разделением модулей определений и модулей реализации. Первые определяют все объекты, экспортируемые из соответствующего модуля реализации. В некоторых случаях, таких, как процедуры и типы, в модуле определений специфицируется только то, что существенно для описания интерфейса, т.е. для пользователя модуля.

Данное сообщение отнюдь не учебник по языку Модула-2. Его цель – дать сжатое и (надеемся) ясное описание. Оно предназначено быть справочным пособием для программистов, реализаторов и авторов руководств и арбитром в сомнительных случаях.

2. СИНТАКСИС

Язык – это бесконечное множество предложений, удовлетворяющих его синтаксису. В случае Модулы-2 предложения называются единицами компиляции. Они представляют собой конечные последовательности символов из конечного словаря. Словарь Модулы-2 состоит из идентификаторов, чисел, строк, операций и ограничителей. Они называются лексическими символами (лексемами) и состоят из последовательности литер.

Для описания синтаксиса используются расширенные формы Бэкуса-Наура (РБНФ). Квадратные скобки [] означают, что заключенная в них сентенциальная форма может отсутствовать, а фигурные скобки {} означают ее повторение (возможно, 0 раз). Синтаксические понятия (нетерминальные символы) обозначаются словами, выражающими их интуитивный смысл. Символы словаря языка (терминальные символы) изображаются словами, написанными прописными буквами (резервированные слова) или цепочками литер, заключенными в кавычки (далее просто цепочки). Синтаксическим правилам (продукциям) предшествует символ $ в начале строки.

3. СЛОВАРЬ И ИЗОБРАЖЕНИЕ

Изображение терминальных символов посредством литер зависит от имеющегося набора литер. В данном сообщении используется набор ASCII; при этом надо иметь в виду следующие лексические правила. Пробелы не должны встречаться внутри символов (исключение составляют цепочки). Пробелы и концы строк игнорируются, если они несущественны для разделения символов.

1. Идентификаторы – последовательности букв и цифр. Первым символом должна быть буква.

```
$      Идентификатор = Буква {Буква | Цифра }.
```

Примеры:

x scan Modula ETH ВзятьСимвол перваяБуква

2. Числа могут быть целыми (возможно, без знака) или действительными. Целое – последовательность цифр. Если за числом следует буква B, оно рассматривается как восьмеричное; если буква H – как шестнадцатеричное; если буква C – число рассматривается как литера с данным (восьмеричным) порядковым номером (и имеющая тип CHAR, см. п.6.1). (* Здесь и далее, если не оговорено особо, имеются в виду ссылки на настоящее сообщение. – Прим. перев.*)

Целое i в диапазоне 0<=i<=MaxInt можно рассматривать как типа INTEGER, так и типа CARDINAL. Если оно лежит в диапазоне MaxInt<=i<=MaxCard, то имеет тип CARDINAL. Для 16-разрядных машин: MaxInt=32767, MaxCard=65535.

Действительное число всегда обычно содержит десятичную точку. Кроме того, оно может содержать порядок. Буква E означает "десять в степени". Действительное число имеет тип REAL.

```
$      Число=Целое | Действительное.
$      Целое = Цифра { Цифра } | ВосьмеричнаяЦифра
$          { ВосьмеричнаяЦифра } ("B"|"C") |
$          Цифра { ШестнадцатеричнаяЦифра } "H".
$      Действительное = Цифра {Цифра}
$          "." {Цифра} [Порядок].
$      Порядок = "E" [ "+" | "-" ] Цифра { Цифра }.
$      ШестнадцатеричнаяЦифра =
$          Цифра |"A"|"B"|"C"|"D"|"E"|"F".
$      Цифра = ВосьмеричнаяЦифра |"8"|"9".
$      ВосьмеричнаяЦифра =
$          "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
```

Примеры:

1980 3764B 7BCH 33C 12.3 45.67E-8

3. Цепочки – последовательности литер, заключенные в кавычки. В качестве кавычек могут использоваться как одиночные кавычки (апострофы), так и двойные кавычки. Однако открывающей и закрывающей кавычкой должна быть одна и та же литера, не встречающаяся в строке. Цепочка литер не может переноситься на другую строку.

```
$      Цепочка = "'" { Литера } "'" | '"' { Литера } '"'.
```

Цепочка, состоящая из n литер, имеет тип (см. п.6.4)

```
ARRAY [0..n-1] OF CHAR
```

Примеры: "MODULA"   "C'est chic!"   'Шлягер "Бразилия"'

4. Операции и ограничители – специальные литеры, пары литер или резервированные слова, перечисленные ниже. В резервированные слова входят только прописные буквы, и они не могут выступать в качестве идентификаторов. Символы # и <> – синонимы, как и символы & и AND, ~ и NOT.

+	=	AND	FOR	QUALIDENT
-	#	ARRAY	FROM	RECORD
*	<	BEGIN	IF	REPEAT
/	>	BY	IMPLEMENTATION	RETURN
:=	<>	CASE	IMPORT	SET
&	<=	CONST	IN	THEN
.	>=	DEFINITION	LOOP	TO
,	..	DIV	MOD	TYPE
;	:	DO	MODULE	UNTIL
(	)	ELSE	NOT	VAR
[	]	ELSIF	OF	WHILE
{	}	END	OR	WITH
^	|	EXIT	POINTER	
~		EXPORT	PROCEDURE	

5. Комментарии – произвольная последовательность литер, заключенная в скобки (* и *). Комментарии могут помещаться между любыми двумя символами программы. Они могут быть вложенными и не влияют на смысл программы.

4. ОПИСАНИЯ И ПРАВИЛА ВИДИМОСТИ

Каждый идентификатор, встречающийся в программе, должен быть описан, если он не является стандартным идентификатором. Последние считаются предописанными и допускают использование в любой части программы. Поэтому они называются проникающими. Описания служат для задания некоторых неизменных свойств объекта, таких, например, как является ли он константой, типом, переменной, процедурой или модулем.

После описания идентификатор используется для ссылки на соответствующий объект. Это возможно только в тех частях программы, которые находятся внутри так называемой области видимости описания. В общем случае область видимости простирается на весь тот блок (процедуру или описание модуля), которому принадлежит данное описание и в котором соответствующий объект локален. Это правило видимости дополняется следующими случаями:

1. Если идентификатор x, определенный описанием D1, используется в другом описании (но не в операторе) D2, то D1 должно текстуально предшествовать D2.
2. Тип T1 может использоваться в описании типа указателя T (см. п.6.7), текстуально предшествующего описанию T1, если как T, так и T1 описаны в одном и том же блоке. Это некоторое ослабление правила 1.
3. Если идентификатор, определенный в модуле M1, экспортируется, в область его видимости включается блок, содержащий M1. Если M1 – единица компиляции (см. гл. 14), то в область видимости включаются все единицы компиляции, импортирующие M1.
4. Использование идентификаторов полей записи (см. п.6.5) допустимо только в обозначениях полей и в операторах присоединения, ссылающихся на переменную этого типа записи.

Идентификатор может быть квалифицирован. В этом случае в качестве приставки к нему используется другой идентификатор, обозначающий модуль (см. гл. 11), в котором определен квалифицируемый идентификатор. Эти идентификаторы разделяются точкой. Ниже перечислены стандартные идентификаторы с указанием пунктов, где они вводятся.

$ КвалИдент = Идентификатор { "." Идентификатор }.

ABS	(10.2)	INCL	(10.2)
BITSET	(6.6)	INTEGER	(6.1)
BOOLEAN	(6.1)	LONGINT	(6.1)
CAP	(10.2)	LONGREAL	(6.1)
CARDINAL	(6.1)	MAX	(10.2)
CHAR	(6.1)	MIN	(10.2)
CHR	(10.2)	NIL	(6.7)
DEC	(10.2)	ODD	(10.2)
EXCL	(10.2)	ORD	(10.2)
FALSE	(6.1)	PROC	(6.8)
FLOAT	(10.2)	REAL	(6.1)
HALT	(10.2)	SIZE	(10.2)
HIGH	(10.2)	TRUE	(6.1)
INC	(10.2)	TRUNC	(10.2)
		VAL	(10.2)

5. ОПИСАНИЯ КОНСТАНТ

Описание константы связывает идентификатор со значением константы.

```
$     ОписаниеКонстанты =
$          Идентификатор "=" КонстВыражение.
$     КонстВыражение = Выражение. (!)
```

Константное выражение – выражение, которое может быть вычислено в процессе чтения программы без ее выполнения. Его операндами являются константы (см. гл. 8). Примеры описаний констант:

```
N = 100
предел=2*N-1
все={0..WordSize-1}
граница=MAX(INTEGER)-N
```

6. ОПИСАНИЯ ТИПОВ

Тип данных определяет множество значений, которое может принимать переменная этого типа, и связывает с этим типом идентификатор. Имеется три разновидности структур: массивы, записи и множества.

```
$     ОписаниеТипа = Идентификатор "=" Тип.
$     Тип = ПростойТип | ТипМассив | ТипЗапись
$         | ТипМножество | ТипУказатель | ТипПроцедура.
$     ПростойТип = КвалИдент | Перечисление
$         | ТипДиапазон.
```

Примеры:

```
Цвет = (красный,зеленый,синий)
Индекс = [1..80]
Карта = ARRAY Индекс OF CHAR
Узел = RECORD ключ:CARDINAL;
           левый,правый:УкДерева
       END
Оттенок = SET OF Цвет
УкДерева = POINTER TO Узел
Функция = PROCEDURE (CARDINAL):CARDINAL
```

6.1. Основные типы

Следующие основные типы предопределены и обозначаются стандартными идентификаторами:

1. **INTEGER** содержит целые числа в диапазоне от **MIN(INTEGER)** до **MAX(INTEGER)**.
2. **CARDINAL** содержит целые числа в диапазоне от 0 до **MAX(CARDINAL)**.

3. **BOOLEAN** содержит истинностные значения **TRUE** и **FALSE**.
4. **CHAR** включает в себя набор литер, предоставляемый вычислительной системой.
5. **REAL** (и **LONGREAL**) обозначает конечное множество действительных чисел.
6. **LONGINT** содержит целые числа в диапазоне от **MIN(LONGINT)** до **MAX(LONGINT)**.

(* В разд. 29 основного текста упоминается также тип LONGCARD. – Прим.перев.*)

6.2. Перечисления

Перечисление представляет собой список идентификаторов, обозначающих значения, образующие тип данных. Эти идентификаторы используются в программе как константы. Они, и только они образуют соответствующий тип данных. Их значения упорядочены, и отношение порядка определяется последовательностью идентификаторов в перечислении. Порядковый номер первого значения равен 0.

```
$     Перечисление = "(" СписИдент ")" .
$     СписИдент = Идентификатор { "," Идентификатор } .
```

Примеры перечислений:

```
(красный, зеленый, синий)
(трефы, бубны, червы, пики)
(Понедельник, Вторник, Среда, Четверг,
 Пятница, Суббота, Воскресенье)
```

6.3. Тип диапазон

Тип T может быть определен как диапазон другого типа T1, основного или перечисления (за исключением **REAL**), путем указания наименьшего и наибольшего значения диапазона.

```
$     ТипДиапазон = [Идентификатор]
$        "[" КонстВыражение ".." КонстВыражение "]" . (!)
```

Первая константа определяет нижнюю границу, и она не должна превышать верхнюю границу. Тип границ T1 называется базовым типом для T, и все операции, применимые к операндам типа T1, применимы также к операндам типа T. Однако значение, присваиваемое переменной типа диапазон, должно лежать внутри заданного отрезка. Базовый тип может задаваться идентификатором, предшествующим указанию границ. Если он опущен и нижняя граница – неотрицательное целое, базовым типом диапазона считается **CARDINAL**; если это отрицательное целое, то **INTEGER**.

Говорят, что тип T1 <u>совместим</u> с типом T0, если либо он описан

как T1=T0, либо как диапазон T0, либо если T0 является диапазоном T1, либо если и T0, и T1 – оба диапазоны одного и того же (базового) типа. Примеры типов диапазон:

```
[0..N-1]
["A".."Z"]
[Понедельник..Пятница]
```

6.4. Тип массив

Массив – структура, состоящая из фиксированного числа компонент одного типа, называемого типом компонент. Элементы массива обозначаются индексами, значения которых принадлежат типу индекса. Описание типа массив определяет тип компонент и тип индекса. Последний должен быть перечислением, типом диапазон или одним из основных типов **BOOLEAN** или **CHAR**.

```
$      ТипМассив = ARRAY ПростойТип {"," ПростойТип} OF Тип.
```

Описание вида

```
ARRAY T1,T2, ... , Tn OF T
```

с n типами индексов T1 ... Tn можно рассматривать как сокращение описания

```
ARRAY T1 OF
  ARRAY T2 OF
    ...
    ARRAY Tn OF T
```

Примеры типов массив:

```
ARRAY [0..N-1] OF CARDINAL
ARRAY [1..10],[1..20] OF [0..99]
ARRAY [-10..+10] OF BOOLEAN
ARRAY ДеньНедели OF Цвет
ARRAY Цвет OF ДеньНедели
```

6.5. Тип запись

Тип запись – структура, состоящая из фиксированного числа компонент, возможно, различных типов. Описание типа записи определяет для каждой компоненты ее тип и идентификатор, обозначающий эту компоненту. Областью видимости этих идентификаторов компонент является само определение записи; кроме того, они присутствуют внутри обозначений (см. п.8.1), ссылающихся на соответствующие компоненты переменных данного типа запись, а также внутри операторов присоединения.


```
$      ТипЗапись = RECORD ПослСписковКомпонент END .
$      ПослСписковКомпонент = СписокКомпонент
$          {";" СписокКомпонент}.
$      СписокКомпонент = [СписИдент ":" Тип|
$            CASE [Идентификатор]":"КвалИдент OF Вариант
$                  {"|" Вариант }                    (!)
$            [ELSE ПослСписковКомпонент] END].
$      Вариант = [СписокМетокВарианта ":"
$                  ПослСписковКомпонент].(!)
$      СписокМетокВарианта = МеткиВарианта
$               {"," МеткиВарианта}.
$      МеткиВарианта=КонстВыражение [".."КонстВыражение].
```

Примеры типов записи:

```
RECORD день:[1..3];
       месяц:[1..12];
       год:[0..2000]
END

RECORD
  имя,фамилия:ARRAY[0..9] OF CHAR;
  возраст:[0..99];
  зарплата:REAL
END

RECORD x,y:T0;
  CASE призн0:Цвет OF
    красный:a:Тк1;b:Тк2|
    зеленый:c:Тз1;d:Тз2|
    синий:e:Тс1;f:Тс2
  END;
  z:T0;
  CASE призн1:BOOLEAN OF
    TRUE:u,v:INTEGER |
    FALSE:r,s:CARDINAL
  END
END
```

Приведенный выше пример содержит два вариантных списка компонент. В первом случае выбор варианта определяется значением компоненты призн0, стоящей в заголовке варианта и называемой дискриминантом или селектором варианта. Во втором случае – дискриминантом призн1.

6.6. Тип множество

Тип множество, определенный как **SET OF T**, содержит все множества значений базового типа T. Базовым типом должен быть либо

диапазон целых между 0 и N-1, либо перечислимый тип (или его диапазон) не более чем с N значениями, где N – небольшая константа, определяемая реализацией (обычно это размер машинного слова или его небольшое кратное).

```
$       ТипМножество = SET OF ПростойТип.
```

Стандартный тип BITSET определяется следующим образом:

```
BITSET=SET OF [0..W-1]
```

Здесь W – константа, определяемая реализацией (обычно это размер машинного слова).

6.7. Тип указатель

Переменные типа указатель P в качестве значений имеют указатели на переменные другого типа T. Говорят, что тип указатель P подчинен T. Значение указателя формируется при обращении к процедуре размещения в модуле управления памятью.

```
$       ТипУказатель = POINTER TO Тип.
```

Помимо значений-указателей, переменная типа указатель может иметь значение NIL, не являющееся указателем ни на какую переменную.

6.8. Тип процедура

Переменная T типа процедура может своим значением иметь процедуру P. Типы формальных параметров P должны совпадать с типами, указанными в списке формальных типов T. Это же относится *и к типу результата в случае процедуры-функции.* (Ограничение: P не может быть локальной или стандартной процедурой.)

```
$       ТипПроцедура = PROCEDURE [СписокФормТипов] .
$       СписокФормТипов = "(" [[VAR] ФормТип
$          {"," [VAR] ФормТип }] ")" [":" КвалИдент].
```

Стандартный тип PROC обозначает процедуру без параметров:

```
PROC = PROCEDURE
```

7. ОПИСАНИЯ ПЕРЕМЕННЫХ

Описание переменной служит для введения переменной и связывания с ней уникального идентификатора, типа данных и структуры. Все

переменные, перечисленные в одном списке, получают одинаковый тип.

```
$       ОписаниеПеременной = СписИдент ":" Тип.
```

Тип данных определяет множество значений, которые может принимать переменная, и операции, которые можно над ней выполнять. Кроме того, он определяет структуру переменной. Примеры описаний переменных (см. примеры в гл. 6):

```
i,j:CARDINAL
  k:INTEGER
p,q:BOOLEAN
  s:BITSET
  F:Функция
  a:ARRAY Индекс OF CARDINAL
  w:ARRAY [0..7]OF
      RECORD ch:CHAR;
              счетч:CARDINAL
      END
```

8. ВЫРАЖЕНИЯ

Выражения – конструкции, дающие правила вычислений для получения значений переменных и формирования новых значений. Выражения состоят из операндов и знаков операций. Для группирования операндов и операций могут использоваться скобки.

8.1. Операнды

За исключением литерных констант, т.е. чисел, цепочек и множеств (см. гл. 5), операнды представляются обозначениями. Обозначение состоит из идентификатора, ссылающегося на обозначаемую константу, переменную или процедуру. Этот идентификатор может быть квалифицирован идентификатором модуля (см. гл. 4 и 11) и за ним могут следовать селекторы, если обозначаемый объект – элемент структуры. Если структура – массив A, то обозначение A[E] означает элемент A, индекс которого – текущее значение выражения E. Тип индекса A должен быть совместим по присваиванию с типом E (см. п.9.1). Обозначение вида A[E1,E2,...,En] означает то же, что и A[E1][E2]...[En].

Если структура – запись R, то R.f обозначает компоненту f записи R. Обозначение P^ используется для переменной, на которую ссылается указатель P.

```
$       Обозначение = КвалИдент {"." Идентификатор |
$                       "[" СписВыражений "]" | "^" }.
$       СписВыражений = Выражение {"," Выражение}.
```

6 Зак. 587

Если обозначаемый объект — переменная, то обозначение ссылается на текущее значение переменной. Если объект — процедура-функция, обозначение без списка параметров ссылается на эту процедуру. Если за ним следует (возможно, пустой) список параметров, обозначение предполагает активацию процедуры и в данной точке используется значение, являющееся результатом ее выполнения, т.е. "возвращаемое" значение. Типы фактических параметров должны соответствовать типам формальных параметров, специфицированным в описании процедуры (см. гл. 10). Примеры обозначений (см. примеры в гл. 7):

k	(INTEGER)
a[i]	(CARDINAL)
w[3].ch	(CHAR)
t^.ключ	(CARDINAL)
t^.левый^.правый	(УказВерш)

8.2. Операции

Синтаксис операций определяет старшинство операций в соответствии с четырьмя классами операций. Операция NOT имеет наивысший приоритет, за ней следуют так называемые операции типа умножения, затем операции типа сложения и, наконец, с низшим приоритетом, операции отношения. Последовательности операций одного приоритета выполняются слева направо.

```
$   Выражение = ПростоеВыражение
$                [Отношение ПростоеВыражение].
$   Отношение = "=" | "#" | "<" | "<="
$               | ">" | ">=" | IN.
$   ПростоеВыражение = ["+"|"-"] Слагаемое
$               {ОперацияТипаСложения Слагаемое}.
$   ОперацияТипаСложения = "+" | "-" | OR.
$   Слагаемое = Множитель
$           {ОперацияТипаУмножения Множитель}.
$   ОперацияТипаУмножения = "*" | "/" | DIV
$               | MOD | AND.
$   Множитель = Число | Цепочка | Множество |
$               Обозначение [ФактическиеПараметры] |
$               "(" Выражение ")" | NOT Множитель.
$   Множество = [КвалИдент]"{"[Элемент{","Элемент}]"}".
$   Элемент = Выражение [".." Выражение]. (!)
$   ФактическиеПараметры = "(" [СписВыражений]")".
```

Имеющиеся операции перечислены ниже в таблице. В некоторых случаях одним и тем же знаком операции обозначаются несколько различных операций. В этих случаях конкретная операция определяется типами операндов.

8.2.1. Арифметические операции

Символ	Операция
+	Сложение
−	Вычитание
*	Умножение
/	Действительное деление
DIV	Целое деление
MOD	Остаток от деления

Эти операции (за исключением /) применимы к операндам типа INTEGER, CARDINAL или их диапазонов. Оба операнда должны быть типа CARDINAL или диапазон с базовым типом CARDINAL, и в этом случае результат имеет тип CARDINAL, или они оба должны иметь тип INTEGER или диапазон базового типа INTEGER, и в этом случае результат имеет тип INTEGER.

Операции +, − и * применимы также к операндам типа REAL. В этом случае оба операнда должны иметь тип REAL и результат имеет тип REAL. Операция деления / применима только к операндам типа REAL. При использовании в качестве одноместной операции "−" означает изменение знака, а "+" − тождественную операцию. Изменение знака применимо только к операндам типа INTEGER и REAL. Операции DIV и MOD определяются следующими правилами:

```
x DIV y равно округленному частному x/y
x MOD y равно остатку от деления x DIV y (для y>0)
x=(x DIV y)*y+(x MOD y)
```

8.2.2. Логические операции

Символ	Операция
OR	Логическое сложение
AND	Логическое умножение
NOT	Отрицание

Эти операции применимы к операндам типа BOOLEAN, и их результат имеет тип BOOLEAN.

```
p OR q  означает "если p, то TRUE, иначе q"
p AND q означает "если p, то q, иначе FALSE"
```

8.2.3. Операции над множествами

Эти операции применимы к операндам любого типа множества, и результат операции имеет тот же тип.

Символ	Операция
+	Объединение множеств
–	Разность множеств
*	Пересечение множеств
/	Симметрическая разность множеств

```
x IN (s1+s2)  эквивалентно  (x IN s1)OR(x IN s2)
x IN (s1-s2)  эквивалентно  (x IN s1)AND NOT(x IN s2)
x IN (s1*s2)  эквивалентно  (x IN s1)AND(x IN s2)
x IN (s1/s2)  эквивалентно  (x IN s1)#(x IN s2)
```

8.2.4. Отношения

Отношения дают логический результат. Отношения порядка применимы к основным типам INTEGER, CARDINAL, BOOLEAN, CHAR, REAL, к перечислениям и диапазонам.

Символ	Отношение
=	Равно
#	Не равно
<	Меньше
<=	Меньше или равно (включение множеств)
>	Больше
>=	Больше или равно (включение множеств)
IN	Содержится в (членство во множестве)

Отношения "=" и "#" применимы, кроме того, к множествам и указателям. В применении к множествам "<=" и ">=" означают (собственное) включение. Отношение IN означает членство в множестве. В выражении вида x IN s выражение s должно быть типа SET OF T, где T – тип (совместимый с) x. Примеры выражений (см. примеры в гл. 7):

```
1980                 (CARDINAL)
k DIV 3              (INTEGER)
NOT p OR q           (BOOLEAN)
(i+j)*(i-j)          (CARDINAL)
s-{8,9,13}           (BITSET)
a[i]+a[j]            (CARDINAL)
a[i+j]*a[i-j]        (CARDINAL)
(0<=k)&(k<100)       (BOOLEAN)
t^.ключ=0            (BOOLEAN)
{13..15}<=s          (BOOLEAN)
i IN {0,5..8,15}     (BOOLEAN)
```


9. ОПЕРАТОРЫ

Операторы означают действия. Имеются простые и структурные операторы. В простые операторы в качестве составных частей не входят другие операторы. Простые операторы – присваивание, вызов процедуры, возврат и выход. Структурные операторы состоят из частей, являющихся операторами. Они используются для организации выполнения последовательности, условий, выбора и повторений.

```
$    Оператор = [Присваивание | ВызовПроцедуры |
$              УсловныйОператор | ОператорВыбора |
$              ЦиклПока | ЦиклДо | БезусловныйЦикл |
$              ЦиклСШагом | ОператорПрисоединения |
$              EXIT | RETURN [Выражение]].
```

Оператор может быть пустым, и в этом случае он соответствует отсутствию каких-либо действий.

9.1. Присваивания

Присваивание служит для замены текущего значения переменной новым значением, определяемым значением выражения. Знаком операции присваивания служит ":=", который читается "присвоить".

```
$    Присваивание = Обозначение ":=" Выражение.
```

Слева от знака присваивания находится переменная. После исполнения присваивания переменная имеет значение, полученное в результате вычисления выражения. Старое значение теряется (затирается). Тип переменной должен быть совместим по присваиванию с типом выражения. Говорят, что типы операндов <u>совместимы по присваиванию</u>, если они либо совместимы, либо оба имеют тип INTEGER или CARDINAL, либо являются диапазонами основных типов INTEGER или CARDINAL.

Цепочка длины n1 может быть присвоена переменной типа цепочка длины n2 > n1. В этом случае цепочка дополняется пустой литерой (0C). Цепочка длины 1 совместима с типом CHAR (!). Примеры присваиваний:

```
i:=k
p:=i=j
j:=log2(i+j)
F:=log2
s:={2,3,5,7,11,13}
a[i]:=(i+j)*(i-j)
t^.ключ:=i
w[i+1].ch:="A"
```

9.2. Вызовы процедур

Вызов процедуры служит для активации процедуры. Вызов процедуры может содержать список фактических параметров, которые подставляются вместо соответствующих формальных параметров, определенных в описании процедуры (см. гл. 10). Это соответствие устанавливается по позициям параметров в списках фактических и формальных параметров соответственно. Имеется два вида параметров: <u>параметры-переменные</u> и <u>параметры-значения</u>.

В случае параметров-переменных фактический параметр должен быть обозначением переменной. Если он обозначает компоненту структурной переменной, селектор вычисляется при подстановке фактического параметра на место формального, т.е. до исполнения процедуры. В случае параметра-значения соответствующий фактический параметр должен быть выражением. Это выражение вычисляется до активации процедуры, и получающееся значение присваивается формальному параметру, который в этом случае представляет собой локальную переменную. Типы соответствующих формальных и фактических параметров должны быть идентичны в случае параметров-переменных и совместимы по присваиванию в случае параметров-значений.

```
$       ВызовПроцедуры =
$               Обозначение [ФактическиеПараметры].
```

Примеры вызовов процедур:

```
Read(i)        (см. гл. 10)
Write(J*2+1,6)
INC(a[i])
```

9.3. Последовательности операторов

Последовательность операторов означает последовательное выполнение действий, определяемых входящими в нее операторами, разделяемых точкой с запятой.

```
$       ПослОператоров = Оператор {";" Оператор}.
```

9.4. Условный оператор

```
$       УсловныйОператор = IF Выражение THEN ПослОператоров
$             {ELSIF Выражение THEN ПослОператоров}
$             [ELSE ПослОператоров] END.
```

Выражения, следующие за символами IF и ELSIF, имеют тип **BOOLEAN**. Они вычисляются в порядке их следования, пока какое-либо из них не даст значения TRUE. В этом случае исполняется соответствующая ему последовательность операторов.

Если присутствует предложение **ELSE**, соответствующая ему последовательность операторов исполняется тогда и только тогда, когда все логические выражения дали значение **FALSE**. Пример:

```
IF(литера>="A")&(литера<="Z")THEN
  ЧтениеИдентификатора
ELSIF(литера>="0")&(литера<=")")THEN ЧтениеЧисла
ELSIF литера='"' THEN ЧтениеЦепочки('"')
ELSIF литера="'" THEN ЧтениеЦепочки("'")
ELSE СпецЛитера
END
```

9.5. Оператор выбора

Оператор выбора определяет выбор и исполнение последовательности операторов в соответствии со значением выражения. Сначала вычисляется выражение выбора, после чего исполняется та последовательность операторов, список меток выбора которой содержит полученное значение. Типом выражения выбора должен быть либо основной тип (за исключением **REAL**), либо перечислимый тип, либо тип диапазон, и все метки должны быть совместимы с этим типом. Метки выбора должны быть константами, и ни одно из значений не может встретиться более одного раза. Если среди меток выбора ни одного из вариантов нет значения выражения, выбирается последовательность операторов, следующая за символом **ELSE**.

```
$       ОператорВыбора = CASE Выражение OF Альтернатива
$            {"|" Альтернатива} [ELSE ПослОператоров] END.
$       Альтернатива = [СписокМетокВарианта ":"
$                 ПослОператоров]. (!)
```

Пример:

```
CASE i OF
  0:p:=p OR q; x:=x+y |
  1:p:=p OR q; x:=x-y |
  2:p:=p AND q; x:=x*y
END
```

9.6. Цикл с условием продолжения

Цикл с условием продолжения определяет повторяющееся исполнение последовательности операторов в зависимости от значения логического выражения. Это выражение вычисляется до очередного исполнения последовательности операторов. Повторение прекращается, как только результатом этого вычисления становится значение **FALSE**.

```
$       ЦиклПока = WHILE Выражение DO ПослОператоров END.
```

Примеры:

```
WHILE j>0 DO
  j:=j DIV 2; i:=i+1
END

WHILE i#j DO
  IF i>j THEN i:=i-j
  ELSE j:=j-i
  END
END

WHILE(t#NIL)&(t^.ключ#i) DO
  t:=t^.левый
END
```

9.7. Цикл с условием окончания

Цикл с условием окончания определяет повторяемое исполнение *последовательности операторов в зависимости* от значения логического выражения. Это выражение вычисляется после каждого исполнения последовательности операторов, и повторение прекращается, как только результатом этого вычисления становится значение TRUE. Таким образом, последовательность операторов исполняется хотя бы один раз.

$ ЦиклДо = REPEAT ПослОператоров UNTIL Выражение.

Пример:

```
REPEAT k:=i MOD j; i:=j; j:=k
UNTIL j=0
```

9.8. Цикл с шагом

Цикл с шагом означает, что последовательность операторов должна исполняться многократно с изменением значения некоторой переменной по прогрессии. Эта переменная называется <u>управляющей переменной</u> цикла с шагом. Она не может быть компонентой структурной переменной, не может импортироваться и быть параметром. Ее значение не должно меняться последовательностью операторов.

$ ЦиклСШагом = FOR Идентификатор ":="
$ Выражение TO Выражение
$ [BY КонстВыражение] DO ПослОператоров END.

Цикл с шагом

```
FOR v:=A TO B BY C DO SS END
```

означает многократное исполнение последовательности операторов SS при том условии, что v принимает значения A,A+C, A+2C, ... , A+nC, где A+nC — последнее значение, не превосходящее B. Переменная v называется управляющей переменной, или параметром цикла, A — начальным значением, B — границей, C — шагом. Значения A и B должны быть совместимы (!) с v; C должно быть константой типа INTEGER или CARDINAL. Если шаг не задан, предполагается, что он равен 1. Примеры:

```
FOR i:=1 TO 80 DO j:=j+a[i] END
FOR i:=80 TO 2 BY -1 DO a[i]:=a[i-1] END
```

9.9. Безусловный цикл

Безусловный цикл определяет многократное исполнение последовательности операторов. Он заканчивается исполнением какого-либо оператора выхода в последовательности операторов.

$ БезусловныйЦикл = LOOP ПослОператоров END.

Пример:

```
LOOP
  IF t1^.ключ>x THEN t2:=t1^.левый; p:=TRUE
  ELSE t2:=t1^.правый; p:=FALSE
  END;
  IF t2=NIL THEN
    EXIT
  END;
  t1:=t2
END
```

Циклы с условием продолжения, условием окончания и шагом могут быть записаны с помощью безусловного цикла, содержащего единственный оператор выхода. Использование этих циклов полезно, поскольку они отражают наиболее часто встречающиеся ситуации, когда завершение зависит либо от единственного условия до или после повторяемой последовательности операторов, либо по достижении границы арифметической прогрессии. В то же время безусловный цикл необходим для записи повторяющихся процессов, в которых завершение заранее не определено. Он также полезен в ситуации, пример которой приведен выше. Операторы выхода содержатся в безусловном цикле контекстуально, а не синтаксически.

9.10. Оператор присоединения

Оператор присоединения определяет переменную запись и последовательность операторов. В операторах этой последовательности квалификация идентификаторов компонент может быть опущена, если они используются для ссылки на переменную, заданную в заголовке оператора присоединения. Если обозначение дает компоненту структурной переменной, селектор вычисляется один раз (до исполнения последовательности операторов). Оператор присоединения открывает новую область видимости.

```
$       ОператорПрисоединения = WITH Обозначение
$               DO ПослОператоров END.
```

Пример:

```
    WITH t^ DO
      ключ:=0; левый:=NIL;правый:=NIL
    END
```

9.11. Операторы выхода и возврата

Оператор возврата состоит из слова **RETURN**, за которым, возможно, следует выражение. Он приводит к завершению процедуры (или тела модуля), а выражение определяет значение, возвращаемое как результат процедуры-функции. Тип этого выражения должен быть совместим по присваиванию с типом результата, определенным в заголовке процедуры (см. гл. 10).

Процедура-функция требует присутствия оператора возврата, задающего значение результата. Таких операторов может быть несколько, но выполняется только один из них. В собственно процедурах под оператором возврата подразумевается конец тела процедуры. Поэтому явное задание оператора возврата рассматривается как дополнительная, возможно, для исключительных ситуаций, точка завершения.

Оператор выхода состоит из слова **EXIT**. Он определяет завершение охватывающего его безусловного цикла и передачу управления на оператор, следующий за этим безусловным циклом (см. п.9.9).

10. ОПИСАНИЯ ПРОЦЕДУР

Описание процедуры состоит из <u>заголовка процедуры</u> и блока, называемого <u>телом процедуры</u>. Заголовок определяет идентификатор процедуры и <u>формальные параметры</u>. Блок содержит описания и операторы. Идентификатор процедуры повторяется в конце описания процедуры.

Имеются два вида процедур, а именно <u>собственно процедура</u> и

<u>процедура-функция</u>. Последняя активируется обозначением функции, входящим в выражение, и вырабатывает результат, являющийся операндом выражения. Собственно процедура активируется вызовом процедуры. Для процедуры-функции в описании после списка параметров указывается тип результата. Ее тело должно содержать оператор RETURN, определяющий результат процедуры-функции.

Все константы, переменные, типы, модули и процедуры, описанные в блоке, составляющем тело процедуры, <u>локальны</u> для процедуры. Значения локальных переменных, включая и определенные в локальных модулях, не определены до входа в процедуру. Поскольку процедуры в свою очередь могут быть описаны как локальные объекты, описания процедур могут быть вложенными. Для каждого описанного объекта можно определить <u>уровень</u> его вложенности. Если объект локален для процедуры уровня k, то его уровень равен k+1. Объекты, описанные в модуле, являющемся единицей компиляции (см. гл.14), имеют по определению уровень 0.

Помимо формальных параметров и локальных объектов, в процедуре доступны также объекты, описанные в окружении процедуры (за исключением объектов, имеющих имена, совпадающие с именами локальных объектов).

Использование идентификатора процедуры в вызове внутри ее описания предполагает рекурсивную активацию процедуры.

```
$       ОписаниеПроцедуры = ЗаголовокПроцедуры ";"
$                  Блок Идентификатор .
$       ЗаголовокПроцедуры = PROCEDURE Идентификатор
$                 [ФормальныеПараметры].
$       Блок = {Описание} [BEGIN ПослОператоров] END.
$       Описание = CONST {ОписаниеКонстанты ";"}|
$            TYPE {ОписаниеТипа ";"}|
$            VAR {ОписаниеПеременной ";" }|
$            ОписаниеПроцедуры ";" | ОписаниеМодуля ";".
```

10.1. Формальные параметры

Формальные параметры – идентификаторы, обозначающие фактические параметры, задаваемые при вызове процедуры. Соответствие между формальными и фактическими параметрами устанавливается при вызове процедуры. Имеется два вида параметров: <u>параметры-значения</u> и <u>параметры-переменные</u>. Разновидность параметра указывается в списке формальных параметров. Параметры-значения представляют собой локальные переменные, которым в качестве начальных значений присваиваются результаты вычисления соответствующих фактических параметров. Параметры-переменные соответствуют фактическим параметрам, являющимся переменными, и они подставляются вместо этих переменных. Параметры-переменные указываются символом VAR, у параметров-значений символа VAR нет.

Формальные параметры локальны для процедуры, т.е. областью их

видимости служит программный текст, составляющий описание процедуры.

```
$       ФормальныеПараметры =
$          "(" [ФПСекция {";" ФПСекция}]")"[":" КвалИдент].
$       ФПСекция = [VAR] СписИдент ":" ФормТип.
$       ФормТип = [ARRAY OF] КвалИдент.
```

Тип каждого формального параметра указывается в списке формальных параметров. В случае параметров-переменных он должен совпадать (!) с типом соответствующего фактического параметра (см. п.9.2 и гл.12, где приведены исключения). В случае параметров-значений формальный тип должен быть совместим по присваиванию с фактическим типом (см. п.9.1). Если параметром является массив, должна использоваться форма

ARRAY OF T

в которой опущена спецификация границ индексов. В таком случае говорят, что параметр является гибким массивом. Тип T должен совпадать с типом элементов фактического массива, а диапазон индексов отображается на целые от 0 до N-1, где N – число элементов. Доступ к формальному массиву только поэлементный, и он может передаваться фактическим параметром, если для соответствующего формального параметра не заданы границы индексов. Процедура-функция без параметров имеет пустой список параметров. Она может быть вызвана обозначением функции также с пустым списком фактических параметров.

> Ограничение: Если формальный параметр определяет тип процедуры, то соответствующий фактический параметр должен быть либо процедурой, описанной на уровне 0, либо переменной (или параметром) типа этой процедуры и не может быть стандартной процедурой.

Примеры описаний процедур:

```
PROCEDURE Read(VAR x:CARDINAL);
  VAR i:CARDINAL; ch:CHAR:
BEGIN i:=0;
  REPEAT ReadChar(ch)
  UNTIL (ch>="0")&(ch<="9");
  REPEAT i:=10*i+(ORD(ch)-ORD("0"));
    ReadChar(ch)
  UNTIL (ch<"0")OR(ch>"9");
  x:=i
END Read
```



```
PROCEDURE Write(x,n:CARDINAL);
  VAR i:CARDINAL;
    buf:ARRAY[1..10]OF CARDINAL;
  BEGIN i:=0;
    REPEAT INC(i); buf[i]:=x MOD 10; x:=x DIV 10
    UNTIL x=0;
    WHILE n>i DO
      WriteChar(" ");DEC(n)
    END;
    REPEAT WriteChar(CHR(buf[i]+ORD("0")));
      DEC(i)
    UNTIL i=0;
 END Write

 PROCEDURE log2(x:CARDINAL):CARDINAL;
   VAR y:CARDINAL; (*предполагается x>0 *)
   BEGIN x:=x-1; y:=0;
     WHILE x>0 DO
       x:=x DIV 2; y:=y+1
     END;
     RETURN y
   END log2
```

10.2. Стандартные процедуры

Стандартные процедуры являются предопределенными. Некоторые из них – встраиваемые процедуры и не могут быть описаны явно, т.е. они применимы к классам типов операндов или же имеют несколько возможных форм списка параметров. Стандартные процедуры перечислены ниже.

ABS(x)	абсолютное значение; тип результата совпадает с типом аргумента.
CAP(ch)	если ch – строчная буква, то соответствующая прописная буква; если ch – прописная буква, она же является и результатом.
CHR(x)	литера с порядковым номером x. CHR(x)=VAL(CHAR,x).
FLOAT(x)	x типа CARDINAL преобразуется в значение типа REAL.
HIGH(a)	верхняя граница индекса массива a.
MAX(T)	максимальное значение типа T (!).
MIN(T)	минимальное значение типа T (!).
ODD(x)	x MOD 2#0.
ORD(x)	порядковый номер (типа CARDINAL) x в множестве значений, определяемом типом T, которому принадлежит x. T – любой перечислимый тип, CHAR, INTEGER или CARDINAL.

SIZE(T)	число единиц памяти, требуемых для переменной типа T, или число единиц памяти, требуемых для переменной T (!).
TRUNC(x)	действительное число x округляется до целого (типа CARDINAL).
VAL(T,x)	значение с порядковым номером x типа T. T — любой перечислимый тип, или CHAR, INTEGER или CARDINAL. Если x типа T, то VAL(T,ORD(x))=x.
DEC(x)	x:=x-1.
DEC(s,n)	x:=x-n.
EXCL(s,i)	s:=s-{i}.
HALT	Прекращение исполнения программы.
INC(x)	x:=x+1.
INC(x,n)	x:=x+n.
INCL(s,i)	s:=s+{i}.

Процедуры INC и DEC применимы также к операндам перечислимых типов и типа CHAR. В этих случаях результатом является элемент, на n элементов предшествующий x или следующий через n элементов после x.

11. МОДУЛИ

Модуль представляет собой набор описаний и последовательность операторов. Они заключаются в скобки MODULE и END. Заголовок модуля состоит из идентификатора модуля и, возможно, нескольких списков импорта и списка экспорта. Первый из них определяет идентификаторы всех объектов, описанных вне модуля, но используемых внутри него (поэтому должны быть импортированы). Список экспорта определяет идентификаторы всех объектов, которые описаны внутри модуля и используются вне его. Следовательно, модуль образует стену вокруг своих локальных объектов, прозрачность которой находится полностью под контролем программиста.

Уровень видимости объектов, локальных для модуля, определяется равным уровню самого модуля. Их можно рассматривать как локальные для процедуры, охватывающей модуль, но с более ограниченной областью видимости.

```
$     ОписаниеМодуля = MODULE Идентификатор [Приоритет]
$         ";" {Импорт} [Экспорт] Блок Идентификатор.
$     Приоритет = "[" КонстВыражение "]".
$     Экспорт = EXPORT [QUALIFIED] СписИдент ";".
$     Импорт = [FROM Идентификатор] IMPORT СписИдент ";".
```

Идентификатор модуля повторяется в конце описания.

Последовательность операторов, составляющая тело модуля, исполняется при вызове процедуры, для которой модуль локален.

Если в процедуре описано несколько модулей, то их тела исполняются в том порядке, в котором модули расположены. Их тела служат для инициализации локальных переменных, и можно считать, что они предшествуют операторной части охватывающей процедуры.

Если идентификатор встречается в списке импорта (экспорта), то соответствующий объект может использоваться внутри (вне) модуля, как если бы скобки модуля отсутствовали. Если, однако, за символом EXPORT следует символ QUALIFIED, то при использовании вне модуля перечисленным идентификаторам должен предшествовать идентификатор модуля. Этот случай называется квалифицированным экспортом и применяется при разработке модулей, которые должны использоваться совместно с другими, заранее неизвестными модулями. Квалифицированный экспорт служит для того, чтобы избежать конфликтов идентификаторов при экспорте из различных модулей (когда имеются в виду различные объекты).

Модуль может иметь несколько списков импорта, которым может предшествовать символ FROM и идентификатор модуля. Результатом предложения FROM является снятие квалификации с импортируемых идентификаторов. Следовательно, внутри модуля они могут использоваться так, как будто они экспортированы обычным, неквалифицированным образом.

Если экспортируется тип запись, то экспортируются и все идентификаторы ее компонент. То же относится и к идентификаторам констант перечислимого типа.

Примеры описаний модулей

Приведенный ниже модуль служит для просмотра текста и копирования его в выходную последовательность литер. Входные литеры последовательно выбираются процедурой ВзятьЛит и выводятся процедурой ВывестиЛит. Литеры представлены в коде ASCII: управляющие литеры, за исключением LF (конец строки) и FS (разделитель файлов), игнорируются. Они оба переводятся в пробел, и при этом устанавливается логическая переменная КонецСтроки или КонецФайла соответственно. Предполагается, что LF предшествует FS.

```
MODULE ВводСтрок;
  IMPORT ВзятьЛит, ВывестиЛит;
  EXPORT Читать, НовСтрока, НовФайл, КонецСтроки,
         КонецФайла, НомСтроки;
  CONST LF=12C; CR=15C; FS=34C;

  VAR НомСтроки:CARDINAL;(*номер строки*)
      ch:CHAR; (*последняя прочитанная литера*)
      КонецФайла, КонецСтроки:BOOLEAN;

  PROCEDURE НовыйФайл;
      BEGIN
```

```
      IF NOT КонецФайла THEN
        REPEAT ВзятьЛит(ch) UNTIL ch=FS;
      END;
      КонецФайла:=FALSE; КонецСтроки:=FALSE;
      НомСтроки:=0
   END НовыйФайл;

   PROCEDURE НовСтрока;
   BEGIN
     IF NOT КонецСтроки THEN
     REPEAT ВзятьЛит(ch) UNTIL ch=LF;
        ВывестиЛит(CR); ВывестиЛит(LF)
     END;
     КонецСтроки:=FALSE; INC(НомерСтроки)
   END НовСтрока;

   PROCEDURE Читать(VAR x:chAR);
     BEGIN
       (*предполагается NOT КонецСтроки и NOT КонецФайла*)
       LOOP ВзятьЛит(ch); ВывестиЛит(ch);
         IF ch>=" " THEN
           x:=ch; EXIT;
         ELSIF ch=LF THEN
           x:=" "; КонецСтроки:=TRUE; EXIT
         ELSIF ch=FS THEN
           x:=" "; КонецСтроки:=TRUE;
           КонецФайла:=TRUE;EXIT
         END
       END
     END Читать;

  BEGIN КонецФайла:=TRUE;КонецСтроки:=TRUE
  END ВводСтрок
```

Следующий пример — модуль, оперирующий с таблицей резервирования дорожек диска и защищающий эту таблицу от недозволенного доступа. Процедура-функция НовДор дает номер свободной дорожки, которая при этом резервируется. Дорожки освобождаются при вызове процедуры ВозврДор.

```
MODULE РезервДор;
  EXPORT НовДор, ВозврДор;
  CONST ЧисДор=1024; (*число дорожек*)
    w=16; (*размер слова*)
    m=ЧисДор DIV w;

  VAR i:CARDINAL;
    Своб:ARRAY[0..m-1]OF BITSET;
```



```
  PROCEDURE НовДор():INTEGER;
    (*резервирует новую дорожку и возвращает ее индекс,
      если свободная дорожка найдена, и -1, если нет*)
    VAR i,j:CARDINAL;Найдена:BOOLEAN;
  BEGIN Найдена:=FALSE; i:=m;
    REPEAT DEC(i);j:=w;
      REPEAT DEC(j);
        IF j IN Своб[i] THEN Найдена:=TRUE END
      UNTIL Нашли OR (j=0)
    UNTIL Нашли OR (i=0);
    IF Нашли THEN EXCL(Своб[i],j);RETURN i*w+j
    ELSE RETURN -1
    END
  END НовДор;

  PROCEDURE ВозврДор(k:CARDINAL);
  BEGIN (*предполагается, что 0<=k<ЧисДор*)
    INCL(Своб[k DIV w],k MOD w)
  END ВозврДор;

 BEGIN (*пометить все дорожки как свободные*)
   FOR i:=0 TO m-1 DO Своб[i]:={0..w-1}END
 END РезервДор
```

12. СИСТЕМНО-ЗАВИСИМЫЕ ВОЗМОЖНОСТИ

Модула-2 предлагает некоторые возможности, необходимые для программирования операций <u>низкого уровня</u>, обращающихся непосредственно к объектам, свойственным данной машине и/или реализации. Сюда входят, например, возможности доступа к устройствам, управляемым машиной, и возможности нарушения правил совместимости типов, наложенных определением языка. Такие возможности следует использовать с крайней осторожностью, и настойчиво рекомендуется ограничить их использование специальными модулями (называемыми модулями низкого уровня). Большинство из них имеет форму типов данных и процедур, импортируемых из стандартного модуля **SYSTEM**. Поэтому модуль низкого уровня явно характеризуется идентификатором **SYSTEM**, стоящим в списке его импорта. (Замечание: Поскольку объекты, импортируемые из **SYSTEM**, подчиняются специальным правилам, этот модуль должен быть доступен компилятору. Поэтому его называют псевдомодулем и он не имеет соответствующего модуля определений (см. гл. 14).)

Возможности, предоставляемые модулем SYSTEM, определяются особенностями реализации. Обычно в их число входят типы **WORD** и **ADDRESS** и процедуры **ADR**, **TSIZE**, **NEWPROCESS**, **TRANSFER** (см. также гл. 13).

Тип **WORD** соответствует адресуемой единице памяти. Над этим

типом не определено никаких операций, кроме присваивания. Однако, если формальный параметр процедуры имеет тип WORD, соответствующий фактический параметр может иметь любой тип, занимающий одно слово памяти в данной реализации. Если формальный параметр имеет тип ARRAY OF WORD, то соответствующий ему фактический параметр может быть любого типа, в частности он может быть типом записи и рассматриваться как массив слов.

Тип ADDRESS определяется как

```
ADDRESS = POINTER TO WORD
```

Он совместим с любым типом указателя, а также с типом CARDINAL. Поэтому к операндам этого типа применимы все операции целой арифметики. Следовательно, тип ADDRESS может использоваться для выполнения адресных вычислений и экспорта результатов в виде указателей. Если формальный параметр имеет тип ADDRESS, соответствующий фактический параметр может иметь любой тип указателя, даже если формальный параметр – это параметр-переменная. Следующий пример примитивного распределителя памяти демонстрирует обычное использование типа ADDRESS.

```
MODULE Storage;
  FROM SYSTEM IMPORT ADDRESS;
  EXPORT Allocate;

  VAR LastUsed:ADDRESS;

  PROCEDURE Allocate(VAR a:ADDRESS;n:CARDINAL);
    BEGIN a:=LastUsed; LastUsed:=LastUsed+n
    END Allocate;

  BEGIN LastUsed:=0
  END Storage;
```

Функция ADR(x) дает адрес памяти переменной x и имеет тип ADDRESS. TSIZE(T) – число единиц памяти, занимаемых любой переменной типа T. TSIZE имеет арифметический тип, зависящий от реализации. (!) Примеры:

```
ADR(LastUsed)  TSIZE(Узел)
```

Помимо экспорта из псевдомодуля SYSTEM, имеются еще две возможности, зависящие от реализации. Первая – использовать тип идентификатора T в качестве имени, обозначающего <u>функцию преобразования типа</u> из типа операнда в тип T. Очевидно, такие функции зависят от представления типа и не включают в себя явных команд преобразования. Вторая нестандартная возможность используется при описании переменной: можно задать абсолютный адрес переменной и игнорировать схему распределения памяти компилятора. Это обеспечивает доступ к памяти, имеющей специальное назначение и фиксированные адреса, такой, например, как регистры устройств машины с вводом-выводом, "отображаемым на память". В этом случае адрес задается целым константным выражением, заключенным в скобки и следующим сразу за идентификатором описываемой переменной. Выбор подходящего типа данных остается за программистом.

13. ПРОЦЕССЫ

Модула-2 разработана прежде всего для реализации на обычной однопроцессорной машине. Для мультипрограммирования она предлагает только некоторые основные возможности, позволяющие определять квазипараллельные процессы и действительную параллельность для периферийных устройств. Под <u>процессами</u> здесь понимаются <u>сопрограммы</u>, исполняемые (одним) процессором поочередно.

13.1. Порождение процессов и передача управления

Новый процесс порождается обращением к

```
PROCEDURE NEWPROCESS(P:PROC; A:ADDRESS;
                     n:CARDINAL; VAR p1:ADDRESS) (!)
```

где P – процедура, образующая процесс,
A – базовый адрес рабочего пространства процесса,
n – размер этого пространства,
p1 – параметр результат.

Новый процесс с программой P и рабочим пространством A размером n присваивается p1. Этот процесс размещается, но не активируется. P должна быть процедурой без параметров, описанной на уровне 0.

Передача управления между двумя процессами осуществляется обращением к

```
PROCEDURE TRANSFER(VAR p1,p2:ADDRESS) (!)
```

При этом приостанавливается выполнение текущего процесса, он присваивается p1, и возобновляется процесс, присвоенный p2. Очевидно, что p2 должен к этому моменту иметь значение, присвоенное либо обращением к NEWPROCESS, либо к TRANSFER. Обе процедуры должны импортироваться. Программа завершается, когда управление доходит до конца процедуры, являющейся телом процесса. (Присваивание p1 происходит после идентификации нового

процесса p2, следовательно, фактические параметры могут совпадать.)

13.2. Процессы устройств и прерывания

Если процесс использует периферийное устройство, то после инициализации обращения к периферийному устройству процессор может быть передан другому процессу, что ведет к параллельному исполнению этого другого процесса с <u>процессом устройства</u>. Обычно завершение операции с устройством сигнализируется прерыванием основного процессора. В терминах Модулы-2 прерывание – это операция передачи управления. Эта передача управления через прерывание (в реализации Модулы-2 на PDP-11) подготавливается одновременно с передачей управления после инициализации устройства. Эти действия объединяются обращением к

```
PROCEDURE IOTRANSFER(VAR p1,p2:ADDRESS;va:CARDINAL) (!)
```

Аналогично **TRANSFER**, вызов **IOTRANSFER** приостанавливает вызывающий процесс устройства, присваивая его p1, возобновляет (передает управление) приостановленный ранее процесс p2 и, кроме того, осуществляет передачу управления по прерыванию, возникающему при завершении работы устройства, присваивая прерванный процесс p2 и возобновляя процесс устройства p1. Здесь va – адрес вектора прерывания, назначенный устройству. Процедура **IOTRANSFER** должна импортироваться, и она рассматривается как зависящая от реализации на PDP-11.

Необходимо, чтобы в некоторых случаях прерывания можно было откладывать (запрещать), например при обращении к переменной, общей для взаимодействующих процессов, или когда требуется выполнить другую операцию с более высоким приоритетом. Поэтому каждый модуль снабжается некоторым уровнем приоритета, и каждое устройство, которое может вызвать прерывание, также снабжается некоторым уровнем приоритета. Исполнение программы может быть прервано тогда и только тогда, когда устройство, вызывающее прерывание, имеет приоритет больший, чем уровень приоритета модуля, содержащего исполняемый в данный момент оператор. В то время как приоритет устройства определяется аппаратурой, уровень приоритета каждого модуля указывается в его заголовке. Если явная спецификация отсутствует, любая процедура имеет такой же уровень, как и вызывающая программа. **IOTRANSFER** можно использовать внутри модулей только со специфицированным приоритетом.

14. ЕДИНИЦЫ КОМПИЛЯЦИИ

Текст, воспринимаемый компилятором как единое целое, называется <u>единицей компиляции</u>. Имеются три разновидности единиц

компиляции: главные модули, модули определений и модули реализации. Главный модуль – основная программа: он состоит из так называемых <u>программных модулей</u>. В частности, он не имеет списка экспорта. Импортируемые объекты определяются в других (раздельно компилируемых) частях программы, которые в свою очередь подразделяются на две единицы компиляции, называемые модулем определений и модулем реализации.

<u>Модуль определений</u> специфицирует имена и свойства объектов, существенные для пользователей, т.е. других модулей, импортирующих из данного. <u>Модуль реализации</u> содержит локальные объекты и операторы, знать о которых пользователь не должен. В частности, модуль определений содержит описания констант, типов и переменных и спецификации заголовков процедур. Соответствующий модуль реализации содержит полные описания процедур и, кроме того, возможно, описания неэкспортируемых объектов. Модули определений и реализаций существуют в паре. И тот и другой могут содержать списки импорта, и все объекты, описанные в модуле определений, доступны в соответствующем модуле реализации без явного импорта.

```
$   МодульОпределений = DEFINITION MODULE Идентификатор
$       ";"{Импорт} {Определение} END
$       Идентификатор ".".(!)
$   Определение = CONST {ОписаниеКонстанты ";"} |
$       TYPE {Идентификатор ["=" Тип]";"}|
$       VAR {ОписаниеПеременной ";"}|
$       ЗаголовокПроцедуры ";".
$   ПрограммныйМодуль = MODULE Идентификатор
$       [Приоритет]";"{Импорт} Блок Идентификатор ".".
$   ЕдиницаКомпиляции = МодульОпределений |
$       [IMPLEMENTATION] ПрограммныйМодуль.
```

Очевидно, что модуль определений представляет собой интерфейс между модулем реализации, с одной стороны, и его пользователями – с другой. Модуль определений содержит те описания, которые нужны модулям-пользователям, и скорее всего никаких других. Следовательно, модуль определений выступает как (расширенный) экспортный список модуля реализации, все описанные объекты которого экспортируются.

Модули определений предполагают использование квалифицированного экспорта. Определение типа может состоять из полной спецификации типа (в этом случае говорят, что экспорт прозрачный), либо оно может состоять только из идентификатора типа. В этом случае полная спецификация должна быть приведена в соответствующем модуле реализации, и в этом случае говорят, что экспорт <u>скрытый</u>. Использующим модулям тип известен только по идентификатору, а все его свойства скрыты. Поэтому процедуры, работающие с операндами этого типа, и в частности с его компонентами, должны определяться в том же модуле реализации,

который скрывает свойства типа. Скрыто экспортировать можно только тип указатель. Ко всем скрытым типам применимы присваивания и проверка на равенство.

Как и в случае локальных модулей, тело модуля реализации действует как инициализатор своих локальных объектов. Перед его исполнением инициализируются импортируемые модули в порядке их перечисления. Если среди модулей имеются циклические ссылки, порядок их инициализации не определен.

ПРИЛОЖЕНИЕ 1

СИНТАКСИС МОДУЛЫ-2

```
Идентификатор = Буква {Буква | Цифра}.
Число=Целое | Действительное.
Целое = Цифра { Цифра } | ВосьмеричнаяЦифра
     { ВосьмеричнаяЦифра } ("B"|"C") |
     Цифра { ШестнадцатеричнаяЦифра } "H".
Действительное = Цифра {Цифра}
     "." {Цифра} [Порядок].
Порядок = "E" [ "+" | "-" ] Цифра { Цифра }.
ШестнадцатеричнаяЦифра =
     Цифра |"A"|"B"|"C"|"D"|"E"|"F".
Цифра = ВосьмеричнаяЦифра |"8"|"9".
ВосьмеричнаяЦифра =
     "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
Цепочка = "'" { Литера } "'" | '"' { Литера } '"'.
КвалИдент = Идентификатор { "." Идентификатор }.
ОписаниеКонстанты =
                 Идентификатор "=" КонстВыражение.
КонстВыражение = Выражение.
ОписаниеТипа = Идентификатор "=" Тип.
Тип = ПростойТип | ТипМассив | ТипЗапись
   | ТипМножество | ТипУказатель | ТипПроцедура.
ПростойТип = КвалИдент | Перечисление
   | ТипДиапазон.
Перечисление = "(" СписИдент ")" .
СписИдент = Идентификатор { "," Идентификатор } .
ТипДиапазон = [Идентификатор]
  "[" КонстВыражение ".." КонстВыражение "]" .
ТипМассив = ARRAY ПростойТип {"," ПростойТип}
            OF Тип.
ТипЗапись = RECORD ПослСписковКомпонент END .
ПослСписковКомпонент = СписокКомпонент
    {";" СписокКомпонент}.
СписокКомпонент = [СписИдент ":" Тип|
      CASE [Идентификатор]":"КвалИдент OF Вариант
                {"|" Вариант }
      [ELSE ПослСписковКомпонент] END].
Вариант = [СписокМетокВарианта ":"
             ПослСписковКомпонент].
```

```
39   СписокМетокВарианта = МеткиВарианта
         {"," МеткиВарианта}.
41   МеткиВарианта=КонстВыражение [".."КонстВыражение].
42   ТипМножество = SET OF ПростойТип.
43   ТипУказатель = POINTER TO Тип.
44   ТипПроцедура = PROCEDURE [СписокФормТипов] .
45   СписокФормТипов = "(" [[VAR] ФормТип
46       {"," [VAR] ФормТип }] ")" [":" КвалИдент].
47   ОписаниеПеременной = СписИдент ":" Тип.
48   Обозначение = КвалИдент {"." Идентификатор |
49                 "[" СписВыражений "]" | "^" }.
50   СписВыражений = Выражение {"," Выражение}.
51   Выражение = ПростоеВыражение
52               [Отношение ПростоеВыражение].
53   Отношение = "=" | "#" | "<" | "<="
54             | ">" | ">=" | IN.
55   ПростоеВыражение = ["+"|"-"] Слагаемое
56               {ОперацияТипаСложения Слагаемое}.
57   ОперацияТипаСложения = "+" | "-" | OR.
58   Слагаемое = Множитель
59           {ОперацияТипаУмножения Множитель}.
60   ОперацияТипаУмножения = "*" | "/" | DIV
61             | MOD | AND.
62   Множитель = Число | Цепочка | Множество |
63             Обозначение [ФактическиеПараметры] |
64             "(" Выражение ")" | NOT Множитель.
65   Множество = [КвалИдент]
66               "{"[Элемент{","Элемент}]"}".
67!  Элемент = Выражение [".." Выражение].
68   ФактическиеПараметры = "(" [СписВыражений]")".
69   Оператор = [Присваивание | ВызовПроцедуры |
70            УсловныйОператор | ОператорВыбора |
71            ЦиклПока | ЦиклДо | БезусловныйЦикл |
72            ЦиклСШагом | ОператорПрисоединения |
73            EXIT | RETURN [Выражение]].
74   Присваивание = Обозначение ":=" Выражение.
75   ВызовПроцедуры =
76            Обозначение [ФактическиеПараметры].
77   ПослОператоров = Оператор {";" Оператор}.
78   УсловныйОператор = IF Выражение THEN ПослОператоров
79          {ELSIF Выражение THEN ПослОператоров}
80          [ELSE ПослОператоров] END.
81   ОператорВыбора = CASE Выражение OF Альтернатива
82        {"|" Альтернатива } [ELSE ПослОператоров] END.
83!  Альтернатива = [СписокМетокВарианта ":"
84            ПослОператоров].
85   ЦиклПока = WHILE Выражение DO ПослОператоров END.
86   ЦиклДо = REPEAT ПослОператоров UNTIL Выражение.
87   ЦиклСШагом = FOR Идентификатор ":="
88       Выражение TO Выражение
89       [BY КонстВыражение] DO ПослОператоров END.
90   БезусловныйЦикл = LOOP ПослОператоров END.
91   ОператорПрисоединения = WITH Обозначение
92           DO ПослОператоров END.
93   ОписаниеПроцедуры = ЗаголовокПроцедуры ";"
94                Блок Идентификатор .
95   ЗаголовокПроцедуры = PROCEDURE Идентификатор
96                [ФормальныеПараметры].
97   Блок = {Описание} [BEGIN ПослОператоров] END.
98   Описание = CONST {ОписаниеКонстанты ";"}|
99        TYPE {ОписаниеТипа ";"}|
100       VAR {ОписаниеПеременной ";" }|
101       ОписаниеПроцедуры ";" | ОписаниеМодуля ";".
102  ФормальныеПараметры =
103     "(" [ФПСекция {";" ФПСекция}]")"[":" КвалИдент].
104  ФПСекция = [VAR] СписИдент ":" ФормТип.
105  ФормТип = [ARRAY OF] КвалИдент.
106  ОписаниеМодуля = MODULE Идентификатор [Приоритет]
107       ";" {Импорт} [Экспорт] Блок Идентификатор.
108  Приоритет = "[" КонстВыражение "]".
109  Экспорт = EXPORT [QUALIFIED] СписИдент ";".
110  Импорт = [FROM Идентификатор] IMPORT СписИдент ";".
111  МодульОпределений = DEFINITION MODULE Идентификатор
112!    ";"{Импорт} {Определение} END
113    Идентификатор ".".
114  Определение = CONST {ОписаниеКонстанты ";"} |
115      TYPE {Идентификатор ["=" Тип]";"}|
116      VAR {ОписаниеПеременной ";"}|
117      ЗаголовокПроцедуры ";".
118  ПрограммныйМодуль = MODULE Идентификатор
119      [Приоритет]";"{Импорт} Блок Идентификатор ".".
120  ЕдиницаКомпиляции = МодульОпределений |
121      [IMPLEMENTATION] ПрограммныйМодуль.
```


Перекрестные ссылки

Альтернатива	-83	82	81					
БезусловныйЦикл	-90	71						
Блок	119	107	-97	94				
Буква	1	1						
Вариант	-37	35	34					
ВосьмеричнаяЦифра	-12	11	4	3				
ВызовПроцедуры	-75	69						
Выражение	88	88	86	85	81	79	78	74
	81	67	67	64	-51	50	50	18
Действительное	-6	2						

ЕдиницаКомпиляции	-120							
ЗаголовокПроцедуры	117	-95	93					
Идентификатор	119	118	115	113	111	110	107	106
	95	94	87	48	34	26	25	25
	19	17	15	15	-1			
Импорт	119	112	-110	107				
КвалИдент	105	103	65	48	46	34	22	-15
КонстВыражение	108	89	41	41	27	22	-18	17
Литера	14	14						
МеткиВарианта	-41	40	39					
Множество	-65	62						
Множитель	64	-62	59	58				
МодульОпределений	120	-110						
Обозначение	91	76	74	63	**-48**			
Оператор	77	77	-69					
ОператорВыбора	-81	70						
ОператорПрисоединения	-91	72						
ОперацияТипаСложения	-57	56						
ОперацияТипаУмножения	-60	59						
Описание	-98	97						
ОписаниеКонстанты	114	98	-16					
ОписаниеМодуля	-106	101						
ОписаниеПеременной	116	100	-47					
ОписаниеПроцедуры	101	-93						
ОписаниеТипа	99	-19						
Определение	-114	112						
Отношение	53	52						
Перечисление	-24	22						
Порядок	-8	7						
ПослОператоров	97	92	90	89	**86**	**85**	**84**	**82**
	80	79	78	-77				
ПослСписковКомпонент	38	36	-31	30				
Приоритет	119	-108	106					
Присваивание	-74	69						
ПрограммныйМодуль	121	-118						
ПростоеВыражение	-55	52	51					
ПростойТип	42	27	27	-22	**20**			
Слагаемое	-58	56	55					
СписВыражений	68	-50	49					
СписИдент	110	109	104	47	33	-25	24	
СписокКомпонент	-33	32	31					
СписокМетокВарианта	83	-39	37					
СписокФормТипов	-45	44						
Тип	115	47	43	33	29	-20	19	
ТипДиапазон	-26	23						
ТипЗапись	-30	20						
ТипМассив	-28	20						
ТипМножество	-42	21						
ТипПроцедура	-44	21						

ТипУказатель	-43	21						
УсловныйОператор	-78	70						
ФактическиеПараметры	76	-68	63					
ФПСекция	-104	103	103					
ФормальныеПараметры	-102	96						
ФормТип	-105	104	46	45				
Целое	-3	2						
Цепочка	62	-14						
ЦиклДо	-86	71						
ЦиклПока	-85	71						
ЦиклСШагом	-87	72						
Цифра	-11	10	8	8	7	6	6	5
	3	3	1					
Число	62	-2						
ШестнадцатеричнаяЦифра	-9	5						
Экспорт	-109	107						
Элемент	-67	66	66					
	14	14						
	53							
	14	14						
	103	68	64	46	24			
	103	68	64	46	24			
	60							
	57	55	8					
	66	50	46	39	27	**25**		
	57	55	8					
	119	113	48	15	7			
	67	41	27					
	60							
	104	103	83	47	46	37	34	33
	87	74						
	119	117	116	115	114	112	110	109
	107	103	101	101	100	99	98	93
	77	**32**						
	53							
	53							
	115	**53**	**19**	**17**				
	54							
	54							
AND	61							
ARRAY	105	**27**						
BEGIN	97							
BY	88							
CASE	81	39						
CONST	114	98						
DEFINITION	111							

ПРИЛОЖЕНИЕ 2

СТАНДАРТНЫЕ ВСПОМОГАТЕЛЬНЫЕ МОДУЛИ

Приводимые далее модули оказались полезными в широком диапазоне применений. В частности, они относятся к вводу и выводу. Модуль Terminal представляет стандартный алфавитно-цифровой терминал, используемый для ввода и вывода. FileSystem предоставляет необходимые операции для создания, чтения, записи, именования и удаления файлов, организованных как потоки литер или слов.

Модули Windows, TextWindows и GraphicWindows образуют иерархию, предназначенную для обслуживания дисплея с высоким разрешением. Два последних используют в качестве базисного модуля Windows. С этими модулями тесно связаны модули CursorMouse и Menu. Первый предполагает наличие указательного устройства, так называемой мыши, для ввода значений координат и отображает его положение на экране дисплея с помощью курсора. Модуль Menu связывает мышь с дисплеем, обеспечивая универсальное средство ввода команд в виде так называемых иерархических меню.

Эти модули представлены здесь в виде модулей определений. Мы подчеркиваем, что они не являются частью языка Модула-2. Различные реализации могут отличаться либо деталями реализации модулей, либо выбором предоставляемых модулей.

```
DEFINITION MODULE Terminal; (*S.E.Knudsen*)
 PROCEDURE Read(VAR ch: CHAR);

 PROCEDURE BusyRead(VAR ch: CHAR);
  (*если клавиша не была нажата, то возвращает 0C*)

 PROCEDURE ReadAgain;
  (*после этого вызова последняя прочитанная литера
    может быть прочитана еще раз вызовом Read*)

 PROCEDURE Write(ch: CHAR);
 PROCEDURE WriteLn; (*завершить строку*)
 PROCEDURE WriteString(s: ARRAY OF CHAR);
END Terminal.
```

```
DEFINITION MODULE FileSystem; (*S.E.Knudsen*)
 FROM SYSTEM IMPORT ADDRESS,WORD;


 TYPE
  Response  = (done,notdone,notsupported,callerror,
         unknownmedium,unknownfile,paramerror,
         toomanyfiles,eom,deviceoff,softparityerror,
         softprotected,softerror,hardparityerror,
         hardprotected,timeout,harderror);
  Command  = (create,open,close,lookup,rename,
         setread,setwrite,setmodify,setopen,doio,
         setpos,getpos,length,setprotect,getprotect,
         setpermanent,getpermanent,getinternal);
  Flag  = (er,ef,rd,wr,ag,bytemode);
  FlagSet = SET OF Flag;


  File  = RECORD res: Response;
        bufa,ela,ina,topa: ADDRESS;
        elodd,inodd,eof: BOOLEAN;
        flags: FlagSet;
        CASE com: Command OF
         create,open,getinternal:
            fileno,versionno: CARDINAL|
         lookup: new: BOOLEAN|
         setpos,getpos,length: highpos,lowpos: CARDINAL|
         setprotect,getprotect: wrprotect: BOOLEAN|
         setpermanent,getpermanent: on: BOOLEAN
        END;
       END;
   (*Процедуры, определяемые файловой системой, могут быть
     сгруппированы следующим образом:
    1. Открытие, закрытие и переименование файлов.
       (Create,Close,Lookup,Rename)
    2. Чтение из файла и запись в файл.
       (SetRead,SetWrite,SetModify,SetOpen,Doio)
    3. Позиционирование файлов.
       (SetPos,GetPos,Length)
    4. Потокоподобная работа с файлами.
       (Reset,Again,ReadWord,
        WriteWord,ReadChar,WriteChar)*)


 PROCEDURE Create(VAR f: File;
                  ИмяУстройства: ARRAY OF CHAR);
  (* создает новый временный (или безымянный) файл
     на указанном устройстве *)
```



```
 PROCEDURE Close(VAR f: File);
  (* завершает операции над файлом f, т.е. разрывает
     связь между переменной f и файловой системой. Тем
     самым временный файл уничтожается, а файл с непу-
     стым именем остается в директории для последую-
     щего использования *)

 PROCEDURE Lookup(VAR f: File;
     filename: ARRAY OF CHAR; new: BOOLEAN);
  (* ищет файл filename. Если файла не существует и
     new=TRUE, то создается новый файл с данным именем *)

 PROCEDURE Rename(VAR f: File; filename: ARRAY OF CHAR);
  (* файл переименуется именем filename, если новое имя
     пусто, то файл f становится временным *)

 PROCEDURE SetRead(VAR f: File);
  (* инициализация файла f на чтение *)

 PROCEDURE SetWrite(VAR f: File);
  (* инициализация файла f на запись *)

 PROCEDURE SetModify(VAR f: File);
  (* инициализация файла f на модификацию *)

 PROCEDURE SetOpen(VAR f: File);
  (* прекращает любую операцию ввода-вывода над файлом f*)

 PROCEDURE Doio(VAR f: File);
  (* используется совместно с SetRead, SetWrite и
     SetModify для последовательного чтения записи и
     модификации файла f *)

 PROCEDURE SetPos(VAR f: File; highpos,lowpos: CARDINAL);
  (* устанавливает текущую позицию файла f на байт
     highpos*2^16 + lowpos *)

 PROCEDURE GetPos(VAR f: File; VAR highpos,lowpos: CARDINAL);
  (* выдает текущую позицию файла f в переменные
     highpos и lowpos*)

 PROCEDURE Length(VAR f: File; VAR highpos,lowpos: CARDINAL);
  (* выдает длину файла f в переменные highpos и lowpos *)

 PROCEDURE Reset(VAR f: File);
  (* выполняет SetOpen и устанавливает позицию файла в
     его начало *)
```

```
PROCEDURE Again(VAR f: File);
 (* после этого вызова, процедуры ReadWord и ReadChar
    прочитают то же самое значение, которое было прочи-
    тано перед этим *)

PROCEDURE ReadWord(VAR f: File; VAR w: WORD);
 (* читает из файла следующее слово *)

PROCEDURE WriteWord(VAR f: File; w: WORD);
 (* пишет в файл следующее слово *)

PROCEDURE ReadChar(VAR f: File; VAR ch: CHAR);
 (* читает из файла следующую литеру *)

PROCEDURE WriteChar(VAR f: File; ch: CHAR);
 (* пишет в файл следующую литеру *)

END FileSystem.


DEFINITION MODULE InOut; (*N.Wirth*)

 CONST EOL = 36C;
 VAR Done: BOOLEAN;
  termCH: CHAR;

 PROCEDURE OpenInput(defext: ARRAY OF CHAR);
  (*запрашивает имя файла и открывает на ввод файл
   "in". Done := "файл успешно открыт". После этого
   вызова последующий ввод происходит из этого файла.
   Если имя заканчивается точкой, то добавляется
   расширение defext *)
 PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
  (*запрашивает имя файла и открывает на ввод файл
   "out". Done := "файл успешно открыт". После этого
   вызова последующий вывод происходит в этот файл *)

 PROCEDURE CloseInput;
  (*закрывает входной файл; возвращает ввод на
   терминал*)

 PROCEDURE CloseOutput;
  (*закрывает выходной файл; возвращает вывод на
   терминал*)

 PROCEDURE Read(VAR ch: CHAR);
  (*Done := NOT in.eof*)
```



```
 PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
  (*чтение цепочки, т.е. последовательности литер, не
   содержащей пробелов и управляющих литер; начальные
   пробелы игнорируются. Ввод прекратится на любой
   литере <= " "; эта литера присваивается переменной
   termCH. Литера DEL используется для забоя литер
   при вводе с терминала*)

 PROCEDURE ReadInt(VAR x: INTEGER);
  (*прочитать строку и преобразовать ее в целое.
   Синтаксис: целое = ["+"|"-"]цифра{цифра}.
   Предшествующие пробелы пропускаются.
   Done := "прочитано число"*)

 PROCEDURE ReadCard(VAR x: CARDINAL);
  (*прочитать строку и преобразовать ее в число типа
   CARDINAL. Синтаксис: целое = цифра{цифра}.
   Предшествующие пробелы пропускаются.
   Done := "прочитано число"*)


 PROCEDURE Write(ch: CHAR);

 PROCEDURE WriteLn; (*завершить строку*)

 PROCEDURE WriteString(s: ARRAY OF CHAR);


 PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
  (*записать целое число x, использовав не менее
   n литер, в файл "out". Если n больше количества
   необходимых позиций, то происходит дополнение
   пробелами слева *)

 PROCEDURE WriteCard(x,n: CARDINAL);

 PROCEDURE WriteOct(x,n: CARDINAL);

 PROCEDURE WriteHex(x,n: CARDINAL);

END InOut.
```

7 Зак. 587

```
DEFINITION MODULE RealInOut; (*N.Wirth*)

 VAR Done: BOOLEAN;

 PROCEDURE ReadReal(VAR x: REAL);
  (*Прочитать действительное число x согласно син-
    таксису:
     ["+"|"-"]цифра{цифра}["." цифра{цифра}]
     ["E"["+"|"-"]цифра[цифра]]

    Done = "число прочитано".
    Учитывается не более 7 цифр, предшествующие нули
    не считаются. Максимальный порядок равен 38.
    Ввод завершается пробелом или управляющей литерой.
    DEL используется для забоя *)

 PROCEDURE WriteReal(x: REAL; n: CARDINAL);
  (*Напечатать x, используя n литер, если требуется
    меньше чем n позиций, то слева вставляются
    пробелы *)

 PROCEDURE WriteRealOct(x: REAL);
  (*Напечатать x в восьмеричной форме с мантиссой и
    порядком *)

END RealInOut.
```

```
DEFINITION MODULE Windows; (*J.Gutknecht*)

 CONST Background = 0; FirstWindow = 1; LastWindow = 8;

 TYPE Window = [Background..LastWindow];
  RestoreProc = PROCEDURE(Window);

 PROCEDURE OpenWindow(VAR u: Window; x,y,w,h: CARDINAL;
     Repaint: RestoreProc; VAR done: BOOLEAN);
  (*Открыть новое окно. Repaint будет
   вызываться для его восстановления*)

 PROCEDURE DrawTitle(u: Window; title: ARRAY OF CHAR);
  (*Вывести заголовок title *)

 PROCEDURE RedefineWindow(u: Window; x,y,w,h: CARDINAL;
     VAR done: BOOLEAN);
  (*Переопределить прямоугольник окна*)
```



```
 PROCEDURE CloseWindow(u:Window); (*Закрыть окно u*)

 PROCEDURE PlaceOnTop(u:Window);
  (*Поместить окно u на верх*)

 PROCEDURE PlaceOnBottom(u:Window);
  (*Поместить окно u в самый низ*)

 PROCEDURE OnTop(u:Window): BOOLEAN;
  (*Окно u - самое верхнее*)

 PROCEDURE UpWindow(x,y: CARDINAL): Window;
  (*Возвращает номер окна или фона,
    соответствующего координатам экрана (x,y)*)

END Windows.
```

```
DEFINITION MODULE TextWindows; (*J.Gutknecht*)

 IMPORT Windows;

 TYPE Window = Windows.Window;
  RestoreProc = Windows.RestoreProc;

 VAR Done: BOOLEAN;
      (*Done="предыдущая операция успешно завершена"*)
  termCH: CHAR; (*завершающая литера*)

 PROCEDURE OpenTextWindow(VAR u: Window;
      x,y,w,h: CARDINAL; name: ARRAY OF CHAR);
  (* Открыть текстовое окно u с
     параметрами x,y,u,w и именем name *)

 PROCEDURE RedefTextWindow(u: Window; x,y,w,h: CARDINAL);
  (* Переопределить текстовое окно *)

 PROCEDURE CloseTextWindow(u: Window);
  (* Закрыть текстовое окно *)

 PROCEDURE AssignFont(u: Window;
      frame,charW,lineH: CARDINAL);
  (*Назначить окну шрифт с номером frame,
   с расстоянием между символами charW и
   между строками lineH *)

 PROCEDURE AssignRestoreProc(u: Window; r: RestoreProc);
  (* Назначить процедуру восстановления окна *)
```

```
PROCEDURE AssignEOWAction(u: Window; r: RestoreProc);
 (*Определить действие, выполняемое по
  достижению конца окна*)

PROCEDURE ScrollUp(u: Window);
 (*Передвинуть текст в окне на одну строку вверх*)

PROCEDURE DrawTitle(u: Window; title: ARRAY OF CHAR);
 (*Напечатать заголовок title *)

PROCEDURE DrawLine(u: Window; line,col: CARDINAL);
 (*col=0:прочертить горизонтальную линию в строке line
  line=0:прочертить вертикальную линию в столбце col*)

PROCEDURE SetCaret(u: Window; on: BOOLEAN);
 (*Управление текстовым курсором: on="курсор на экране"*)

PROCEDURE Invert(u: Window; on: BOOLEAN);
 (*Инвертирование изображения в окне:
   on="изображение негативное"*)

PROCEDURE IdentifyPos(u: Window; x,y: CARDINAL;
     VAR line,col:CARDINAL);
 (*Определить координаты в окне (line,col) точки
  экрана с координатами x,y*)

PROCEDURE GetPos(u: Window; VAR line,col: CARDINAL);
 (*Получить текущую позицию в окне*)

PROCEDURE SetPos(u: Window; line,col: CARDINAL);
 (*Установить текущую позицию в окне*)

PROCEDURE ReadString(u: Window; a: ARRAY OF CHAR);

PROCEDURE ReadCard(u: Window; VAR x: CARDINAL);

PROCEDURE ReadInt(u: Window; VAR x: INTEGER);

PROCEDURE Write(u: Window; ch: CHAR);
 (*Напечатать литеру в текущей позиции.
  Управляющие литеры BS,LF,FF,CR,CAN,EOL и DEL
  соответствующим образом интерпретируются*)
PROCEDURE WriteLn(u: Window);
PROCEDURE WriteString(u: Window; a: ARRAY OF CHAR);
PROCEDURE WriteCard(u: Window; x,n: CARDINAL);
PROCEDURE WriteInt(u: Window; x: INTEGER; n: CARDINAL);
PROCEDURE WriteOct(u: Window; x,n: CARDINAL);

END TextWindows.
```



```
DEFINITION MODULE GraphicWindows; (*E.Kohen*)

 IMPORT Windows;

 TYPE Window = Windows.Window;
  RestoreProc = Windows.RestoreProc;
  Mode = (replace,paint,invert,erase);

 VAR Done: BOOLEAN;
      (*Done="предыдущая операция успешно завершена"*)

 PROCEDURE OpenGraphicWindow(VAR u:Window;
      x,y,w,h: CARDINAL; name: ARRAY OF CHAR;
      Repaint: RestoreProc);
  (*Открыть графическое окно. Написать заголовок name,
    если он не пустой.
    Repaint - процедура перерисовки окна *)

 PROCEDURE RedefGraphicWindow(u: Window;
      x,y,w,h: CARDINAL);
  (*Переопределить прямоугольник окна u*)

 PROCEDURE Clear(u: Window);
  (*Очистить окно u*)

 PROCEDURE CloseGraphicWindow(u: Window);
  (*Закрыть окно*)

 PROCEDURE SetMode(u: Window; m: Mode);
  (*Установить режим окна*)

 PROCEDURE Dot(u: Window; x,y: CARDINAL);
  (*Поставить точку с координатами (x,y)*)

 PROCEDURE SetPen(u: Window; x,y: CARDINAL);
  (*Установить перо в точку (x,y) окна u*)

 PROCEDURE TurnTo(u: Window; angle: INTEGER);
  (*Установить текущее направление в окне u
    на угол angle *)

 PROCEDURE Turn(u: Window; angle: INTEGER);
  (*Повернуть текущее направление окна на угол angle*)

 PROCEDURE Move(u: Window; d: CARDINAL);
  (*Передвинуть перо в текущем направлении на расстояние d*)

 PROCEDURE MoveTo(u: Window; x,y: CARDINAL);
  (*Передвинуть перо из текущего положения в точку (x,y)*)
```

```
 PROCEDURE Circle(u: Window; x,y,r: CARDINAL);
  (*Построить окружность с радиусом r и координатами (x,y)*)

 PROCEDURE Area(u: Window; c: CARDINAL; x,y,w,h: CARDINAL);
  (*Закрасить прямоугольник с координатами x,y
    и размерами w,h цветом c *)

 PROCEDURE CopyArea(u: Window; sx,sy,dx,dy,dw,dh: CARDINAL);
  (*Скопировать прямоугольную область с координатами (sx,sy)
   в область с координатами (dx,dy), шириной dw и высотой dh*)

 PROCEDURE Write(u: Window; ch: CHAR);

 PROCEDURE WriteString(u: Window; s: ARRAY OF CHAR);

 PROCEDURE IdentifyPos(VAR u: Window; VAR x,y: CARDINAL);

END GraphicWindows.
```

```
DEFINITION MODULE CursorMouse; (*J.Gutknecht, 17.11.83*)
 CONST ML = 15; MM = 14; MR = 13;

 TYPE
  Pattern = RECORD
   height: CARDINAL;
   raster: ARRAY [0..15] OF BITSET
  END;

  ReadProc = PROCEDURE(VAR BITSET,VAR CARDINAL,VAR CARDINAL);

 PROCEDURE SetMouse(x,y: CARDINAL);
  (*Установить мышь в точку (x,y)*)

 PROCEDURE GetMouse(VAR s: BITSET; VAR x,y: CARDINAL);
  (*Получить текущее состояние мыши
   ML IN s = "нажата левая кнопка мыши";
   MM IN s = "нажата средняя кнопка мыши";
   MR IN s = "нажата правая кнопка мыши"; *)

 PROCEDURE ReadMouse(VAR s: BITSET; VAR x,y: CARDINAL);
  (*Переназначаемая процедура для имитации мыши*)

 PROCEDURE Assign(p: ReadProc);
  (*Переназначить процедуру ReadMouse*)

 PROCEDURE MoveCursor(x,y: CARDINAL);
  (*Передвинуть курсор в заданную позицию*)
```



```
 PROCEDURE EraseCursor;
  (*Убрать курсор с экрана*)

 PROCEDURE SetPattern(VAR p: Pattern);
  (*Установить собственный шаблон курсора*)

 PROCEDURE ResetPattern;
  (*Установить стандартный шаблон курсора*)

END CursorMouse.
```

```
DEFINITION MODULE Menu; (*J.Gutknecht,6.9.83*)

 PROCEDURE ShowMenu(X,Y: CARDINAL;
     VAR menu: ARRAY OF CHAR; VAR cmd: CARDINAL);
  (*menu = заголов{"|"элемент}.
   элемент = имя["("меню")"].
   имя = {литера}.
   заголов = имя.
   Непечатаемые литеры и превышающие максимальную длину
   имени игнорируются. Входное значение переменной cmd
   указывает на те команды, которые будут первоначально
   выбраны. Последовательность выбранных элементов
   возвращается в виде восьмеричных цифр значения
   переменной cmd справа налево.*)

END Menu.
```

```
DEFINITION MODULE Storage; (*SEK 5.10.80*)
 FROM SYSTEM IMPORT ADDRESS;

 PROCEDURE ALLOCATE(VAR a: ADDRESS; size: CARDINAL);
  (*Выделяет участок памяти указанного размера size и
    возвращает его адрес в a. Если память не может
    быть выделена, то программа завершается*)

 PROCEDURE DEALLOCATE(VAR a: ADDRESS; size: CARDINAL);
  (*Освобождает область памяти с адресом a и заданным
   размером size*)

 PROCEDURE Available(size: CARDINAL): BOOLEAN;
  (*Возвращает TRUE, если можно выделить
   "размер" слов процедурой ALLOCATE.*)

END Storage.
```

```
DEFINITION MODULE MathLib0;
 (*Стандартные функции: J.Waldvogel/N.Wirth,10.12.80*)

  PROCEDURE sqrt(x: REAL): REAL;

  PROCEDURE exp(x: REAL): REAL;

  PROCEDURE ln(x: REAL): REAL;

  PROCEDURE sin(x: REAL): REAL;

  PROCEDURE cos(x: REAL): REAL;

  PROCEDURE arctan(x: REAL): REAL;

  PROCEDURE real(x: INTEGER): REAL;

  PROCEDURE entier(x: REAL): INTEGER;

END MathLib0.
```

ПРИЛОЖЕНИЕ 3

ТАБЛИЦА ЛИТЕР КОДА ASCII

	0	20	40	60	100	120	140	160
0	nul	dle		0	@	P		p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
10	bs	can	(	8	H	X	h	x
11	ht	em	)	9	I	Y	i	y
12	lf	sub	*	:	J	Z	j	z
13	vt	esc	+	;	K	[	k	{
14	ff	fs	,	<	L	\	l	|
15	cr	gs	—	=	M	]	m	}
16	so	rs	.	>	N	^	n	~
17	si	us	/	?	O	_	o	del

Литеры управления курсором

bs	пробел назад
ht	горизонтальная табуляция
lf	перевод строки
vt	вертикальная табуляция
ff	перевод формата
cr	возврат каретки

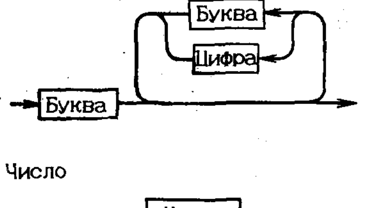
Литеры-разделители

fs	разделитель файлов
gs	разделитель групп
rs	разделитель записей
us	разделитель единиц

ПРИЛОЖЕНИЕ 4

СИНТАКСИЧЕСКИЕ ДИАГРАММЫ МОДУЛЫ-2

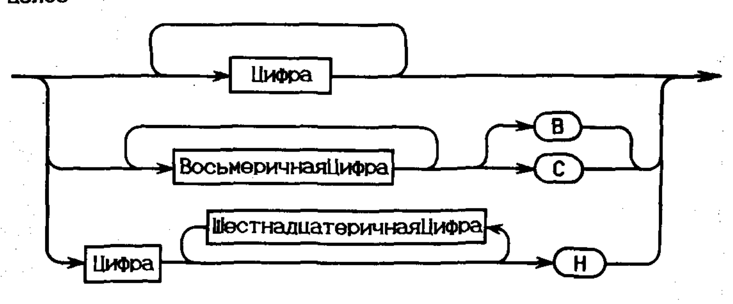
Идентификатор



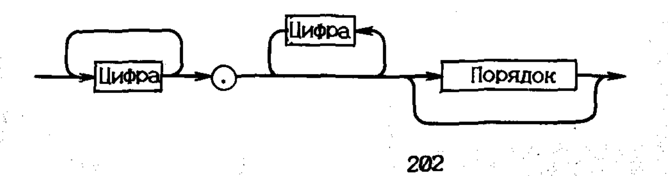
Число



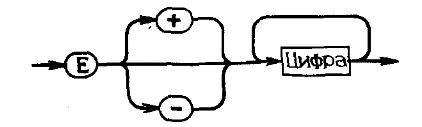
Целое



Действительное



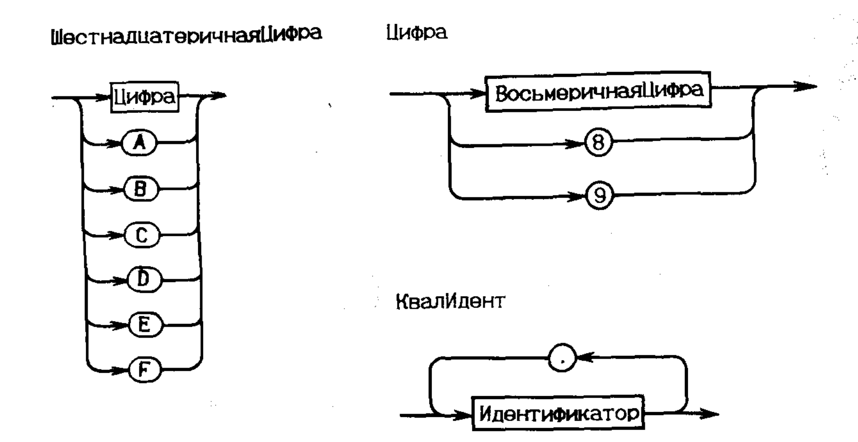
Порядок



ШестнадцатеричнаяЦифра

Цифра

КвалИдент

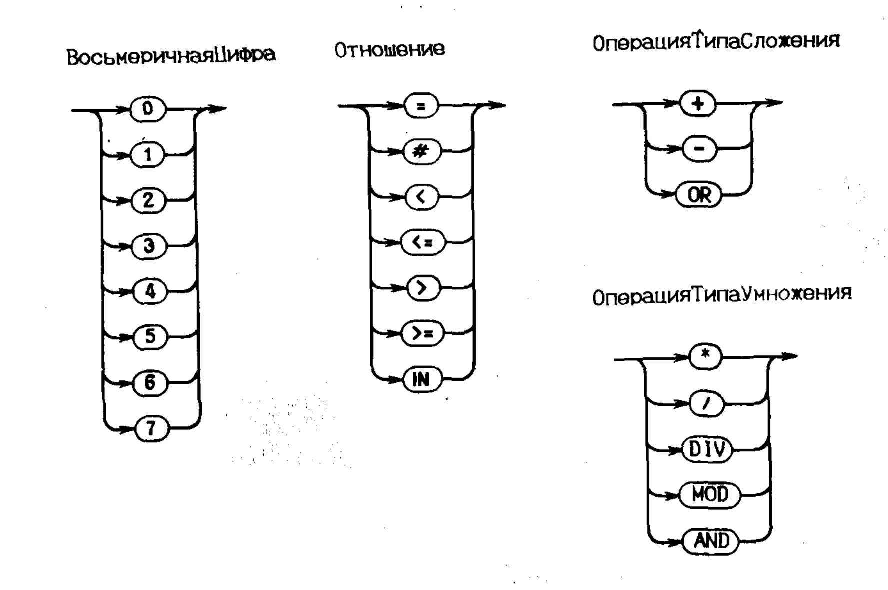


ВосьмеричнаяЦифра

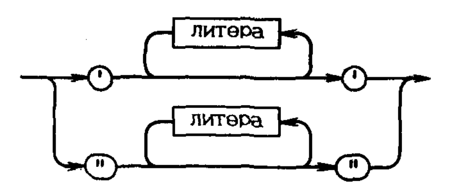
Отношение

ОперацияТипаСложения

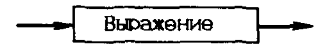
ОперацияТипаУмножения



Цепочка



КонстВыражение



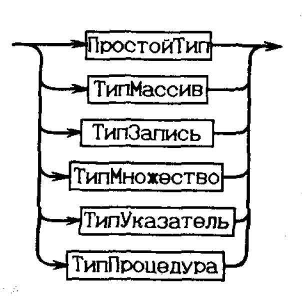
ОписаниеТипа



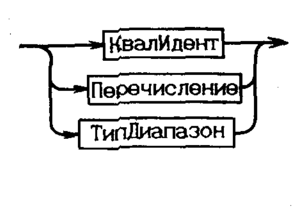
Перечисление



Тип



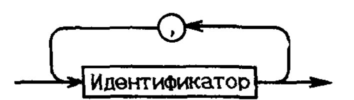
ПростойТип



ОписаниеКонстанты



СписИдент



ТипДиапазон



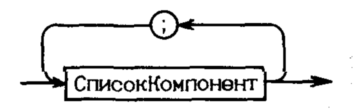
ТипМассив



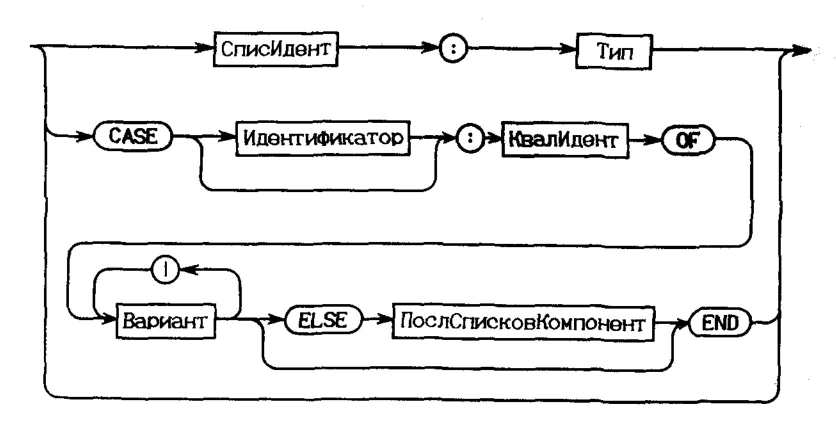
ТипЗапись



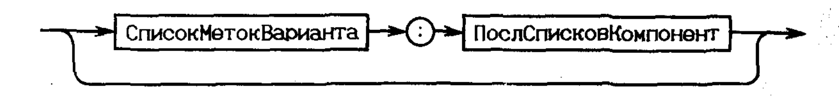
ПослСписковКомпонент



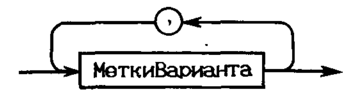
СписокКомпонент



Вариант



СписокМетокВарианта



МеткиВарианта



ТипМножество



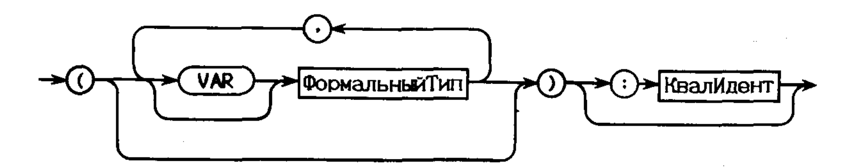
ТипУказатель



ТипПроцедура



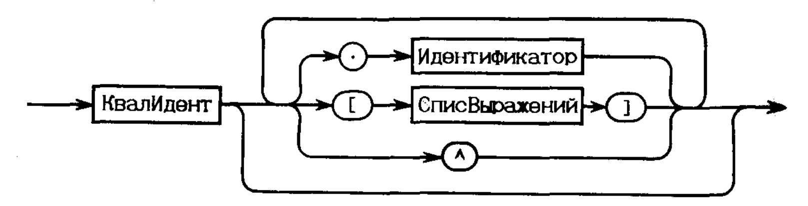
СписокФормТипов



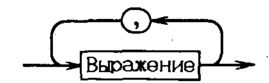
ОписаниеПеременной



Обозначение



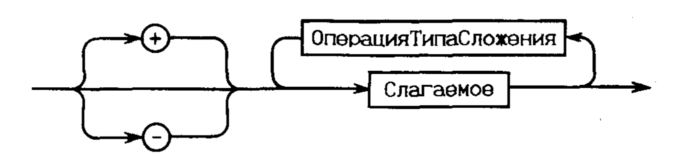
СписВыражений



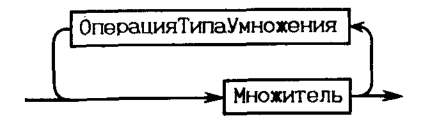
Выражение



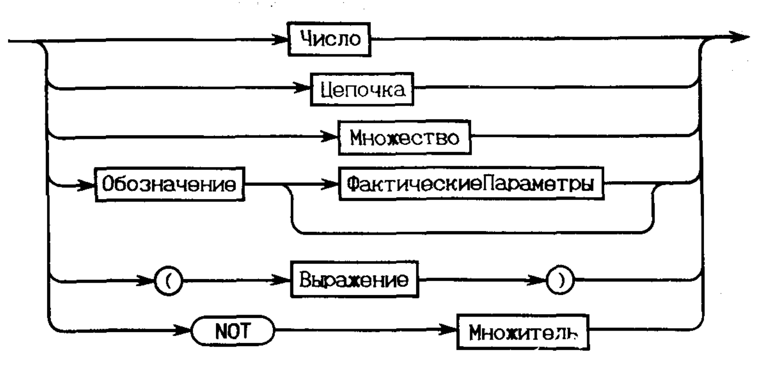
ПростоеВыражение



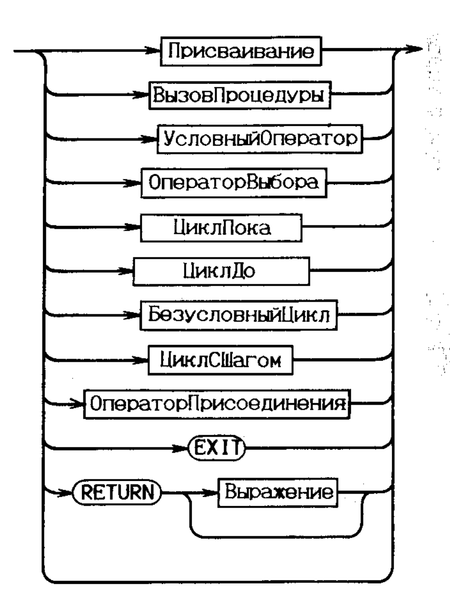
Слагаемое



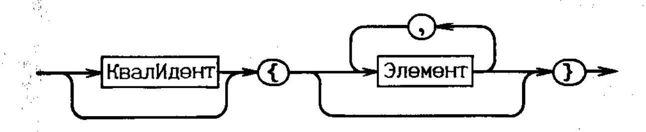
Множитель



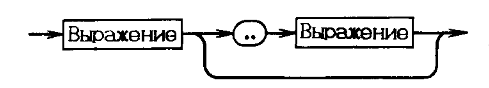
Оператор



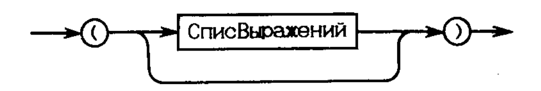
Множество



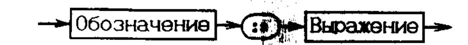
Элемент



ФактическиеПараметры



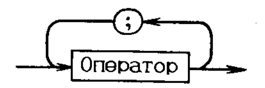
Присваивание



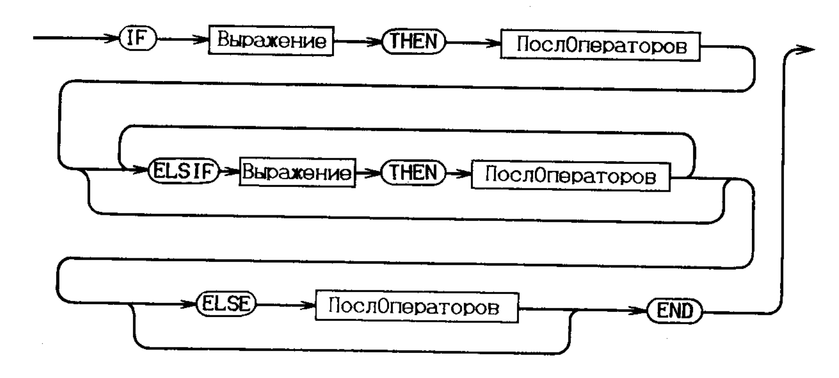
ВызовПроцедуры



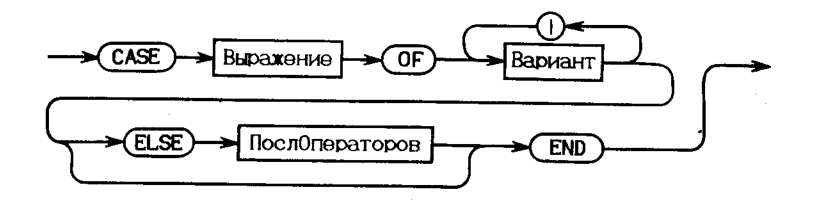
ПослОператоров



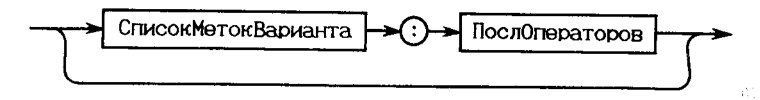
УсловныйОператор



ОператорВыбора



Вариант



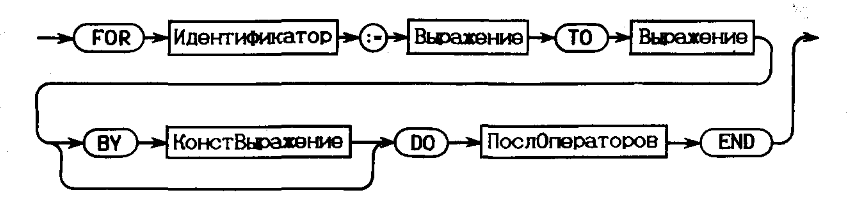
ЦиклПока



ЦиклДо

REPEAT → ПослОператоров → UNTIL → Выражение

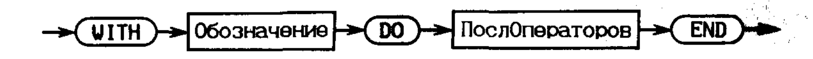
ЦиклСШагом



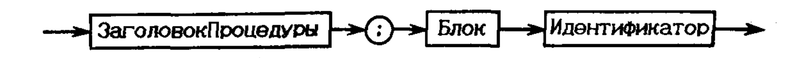
БезусловныйЦикл



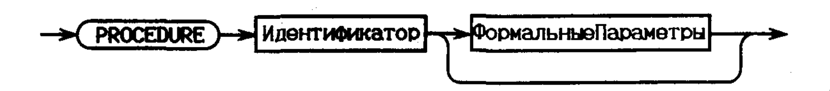
ОператорПрисоединения



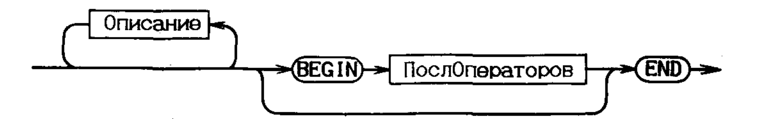
ОписаниеПроцедуры



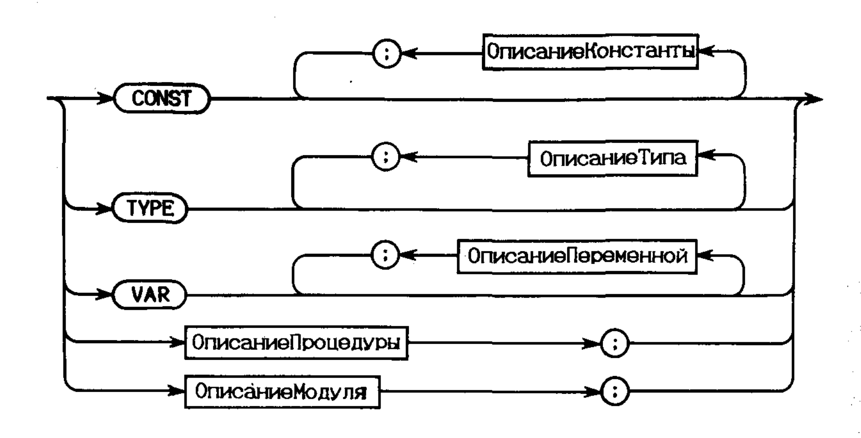
ЗаголовокПроцедуры



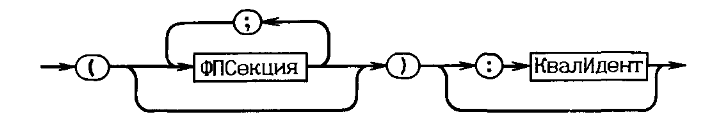
Блок



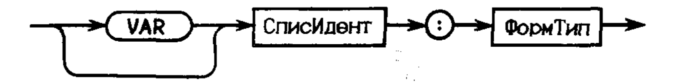
Описание



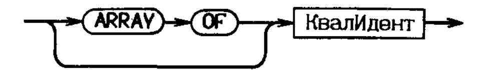
ФормальныеПараметры



ФПСекция



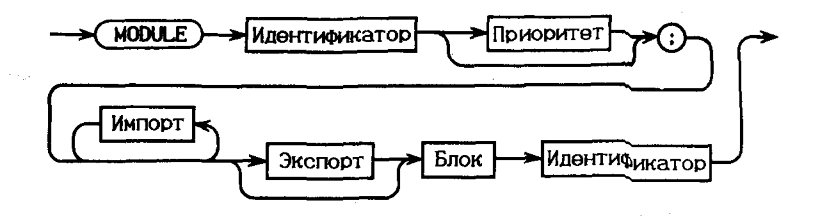
ФормТип



Приоритет



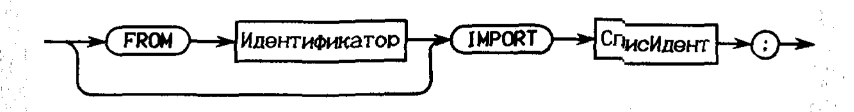
ОписаниеМодуля



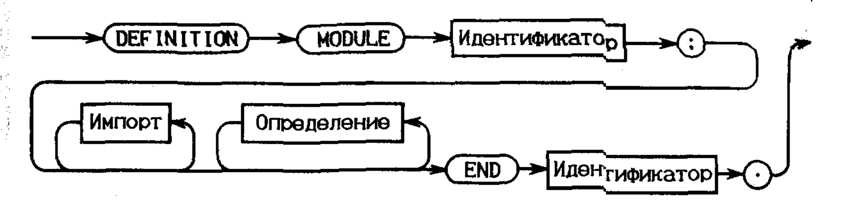
Экспорт



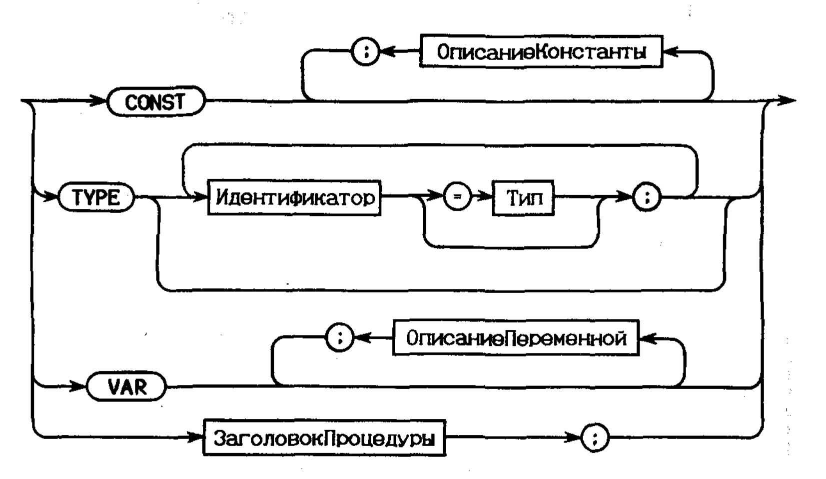
Импорт



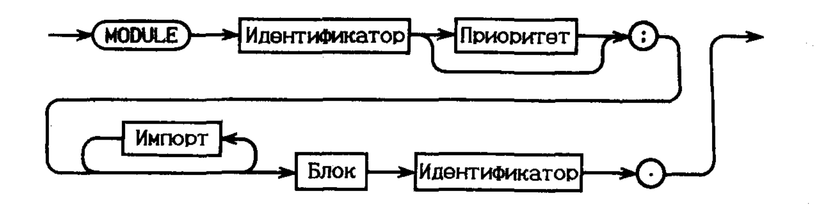
МодульОпределений



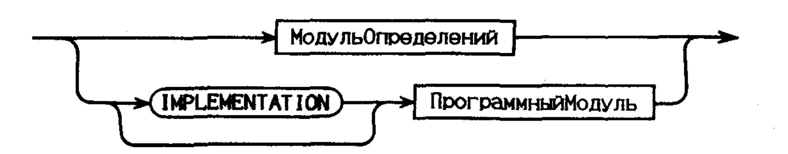
Определение



ПрограммныйМодуль



ЕдиницаКомпиляции



ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

ОГЛАВЛЕНИЕ

Уважаемый читатель!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, издательство "Мир".

Учебное издание

Никлаус Вирт

Программирование на языке Модула-2

Ст. научный редактор И.А. Маховая

Художественный редактор В.И. Шаповалов

Художник Н.Я. Вовк

Корректор С.С. Суставова

ИБ № 5811

Подписано к печати 24.04.87. Формат 60 Х 88 1/16. Бумага офсетная № 1. Печать офсетная. Гарнитура специализированная. Объем 7,00 бум. л. Усл. печ. л. 13,72. Усл. кр.-отт. 14,23. Уч. -изд. л. 13,17. Изд. № 1/4706. Тираж 20 000 экз. Зак № 587. Цена 1 руб.

Издательство "Мир"
129820, ГСП, Москва, И-110, 1-й Рижский пер., 2

Набрано в ВЦ АН СССР на персональной ЭВМ Labtam с печатающим устройством P1350 фирмы Toshiba

Московская типография №4 Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли,
129041, Москва, Б. Переяславская, 46.